1990-06

Multiprocessor scheduling for hard real-time software

Hsu, Liangchuan.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

MULTIPROCESSOR SCHEDULING
FOR HARD REAL-TIME SOFTWARE

by

Liangchuan Hsu

June, 1990

Thesis Advisor: Luqi

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): 52	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
MULTIPROCESSOR SCHEDULING FOR HARD REAL-TIME SOFTWARE

12. PERSONAL AUTHOR(S)
Liangchuan Hsu

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM TO	June 1990	112

16. SUPPLEMENTARY NOTATION: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	RAPID PROTOTYPING, MULTIPROCESSOR, MULTISCHEDULE, HEURISTIC, COST FUNCTION

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report builds upon work previously done in the development of the Computer Aided Prototyping System (CAPS) and the Prototype System Description Language (PDSDL) and presents a conceptual design for the pioneer prototype of the static scheduler for multiprocessors which are part of the CAPS execution support system. The design of hard real-time systems is gaining importance in the software engineering field as real-world processes are becoming automated. This increase in automation needs the advancement of software design technolcgy to meet the design requirements for these hard real-time systems. The CAPS and PSDL are tools being developed to aid the software designer in the rapid prototyping of hard real-time systems. Scheduling PSDL operators in multiprocessor systems to meet the timing constraints is the main part of this thesis. Implementation of the conceptual design will be the basis for further work in this area.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Luqi	22b. TELEPHONE (Include Area Code): (408) 646-2735; 22c. OFFICE SYMBOL: 52Lq

Approved for public release; distribution is unlimited.

MULTIPROCESSOR SCHEDULING FOR HARD REAL-TIME SOFTWARE

by

Liangchuan Hsu
Captain, ROC Army
B.S., Chungcheng Institute of Technology, 1986

Submitted in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1990

Author: [redacted]
Liangchuan Hsu

Approved by: [redacted]
Luqi, Thesis Advisor

[redacted]
Valdis Berzins, Second Reader

[redacted]
Robert B. McGhee, Chairman, Department of Computer Science

ABSTRACT

This thesis builds upon work previously done in the development of the Computer Aided Prototyping System (CAPS) and the Prototype System Description Language (PSDL) and presents a conceptual design for the pioneer prototype of the static scheduler for multiprocessors which are part of the CAPS execution support system. The design of hard real-time systems is gaining importance in the software engineering field as real-world processes are becoming automated. This increase in automation needs the advancement of software design technology to meet the design requirements for these hard real-time systems. The CAPS and PSDL are tools being developed to aid the software designer in the rapid prototyping of hard real-time systems. Scheduling PSDL operators in multiprocessor systems to meet the timing constraints is the main part of this thesis. Implementation of the conceptual design will be the basis for further work in this area.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

Real-time systems are widely applied in many fields. There are two types of real-time systems, namely, soft real-time systems and hard real-time systems. In soft real-time systems, tasks have performance goals, but they are not constrained to finish by specific times. On the other hand, hard real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced. Examples of this type of real-time system are command and control systems, process control systems, flight control systems, space shuttle avionics, systems such as the space station, space-based defense systems such as SDI, and large command and control systems [Ref. 39: P. 1].

Most of the hard real-time computer systems are special-purpose and complex, require a high degree of fault tolerance, and are typically *embedded* in a larger system [Ref. 39: P. 1]. Simply stated, an *embedded* computer is one that is part of a larger system, such as a guidance computer on a missile, a process controller, a business communications network, or even a microprocessor used to control an automobile engine or a microwave oven. An embedded computer system may be anything from a single microcomputer to a network of large computers. In general, embedded systems are large and

have similar requirements for parallel processing, real-time control, and high reliability [Ref. 3: P. 3].

Typically, a real-time system consists of a controlling system and a controlled system. The controlled system can be viewed as the environment with which the computer interacts. Adaptability is particularly important for real-time systems because if a task's deadlines can be met only under a restricted system state/configuration, reliability and performance may be compromised.

In summary, real-time systems differ from traditional systems in that deadlines or other explicit timing constraints are attached to tasks, the systems are in a position to make compromises, and faults, including timing faults, may cause catastrophic consequences. This implies that, unlike many systems where there is a separation between correctness and performance, in real-time systems correctness and performance are very tightly interrelated. Thus, real-time systems solve the problem of missing deadlines in ways specific to the requirements of the target application.

B. SOFTWARE ENGINEERING AND RAPID PROTOTYPING

1. Software Engineering

Software Engineering is the application of science and mathematics to the problem of making computers useful to people by means of *software*. *Software* is the entire set of documentation, operating procedures test case, and programs associated with a computer-bases system [Ref. 2]. It is not just programming. An

abstraction is a simplified view of a system containing only the details important for a particular purpose. Just as experience can help us solve the similar things in our lives, abstractions provide the templates for large scale projects.

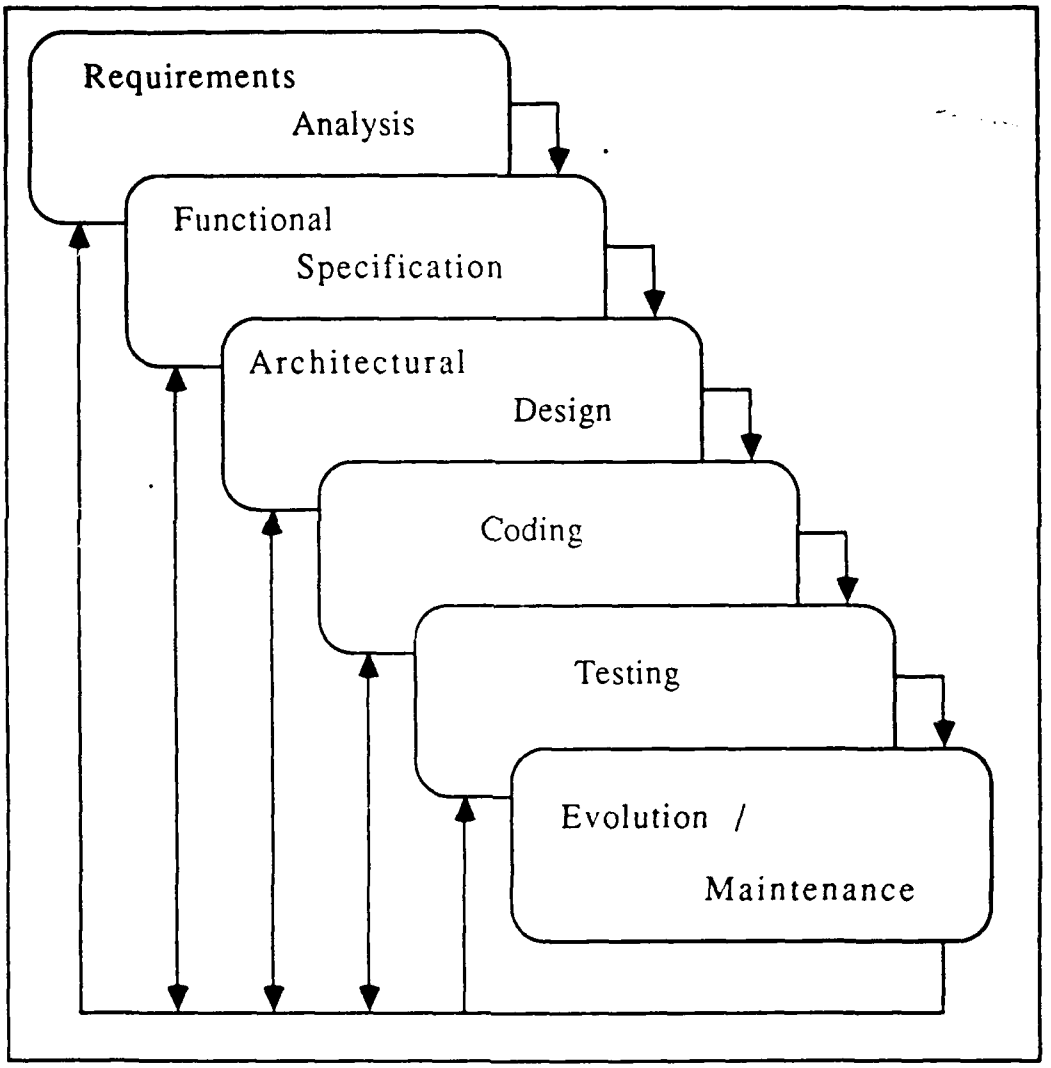


Figure 1 Traditional Software Life Cycle

2. Traditional Software Cycle

Figure 1 on page 3 illustrates the traditional life cycle paradigm for software engineering. Sometimes it is called the "waterfall model" [Ref. 36: P. 20]. Requirements analysis is the process of determining and documenting the customer's needs and constraints. Functional specification is the process of proposing and formalizing a systems interface for meeting the customer's needs. Architectural design is the process of decomposing the system into modules and defining internal interfaces. Implementation is the process of producing an executable program unit for each module, which can be divided into two parts. Coding translates the module into a machine-executable form, and testing assures that defined input produces actual results. Evolution is the process of adapting the system to the changing needs of the customer. Figure 2 on page 5 illustrates the cost distribution of software development [Ref. 2].

Figure 3 on page 6 contains three generic phases of software engineering. The definition phase focuses on *what*. The development phase focuses on *how*. The maintenance phase focuses on *change* that is associated with error correction, adaptations required as the software's environment evolves, and modifications due to enhancements brought about by changing customer requirements. [Ref. 36: PP. 27-28]

3. Rapid Prototyping

A *prototype* is an executable model or a pilot version of the intended system [Ref. 23: P. 1409]. A *prototype* is usually

a partial representation of the intended system, used as an aid in analysis and design rather than as production software. The purpose of a prototype is to provide answers to questions about the requirements and the properties of the proposed system. A *prototype* does not have to be complete, reliable, or efficient. However, a prototype must have the following properties :

(1) be traceable to its requirements,

(2) be easy to modify, and

(3) be easy to read and analyze.

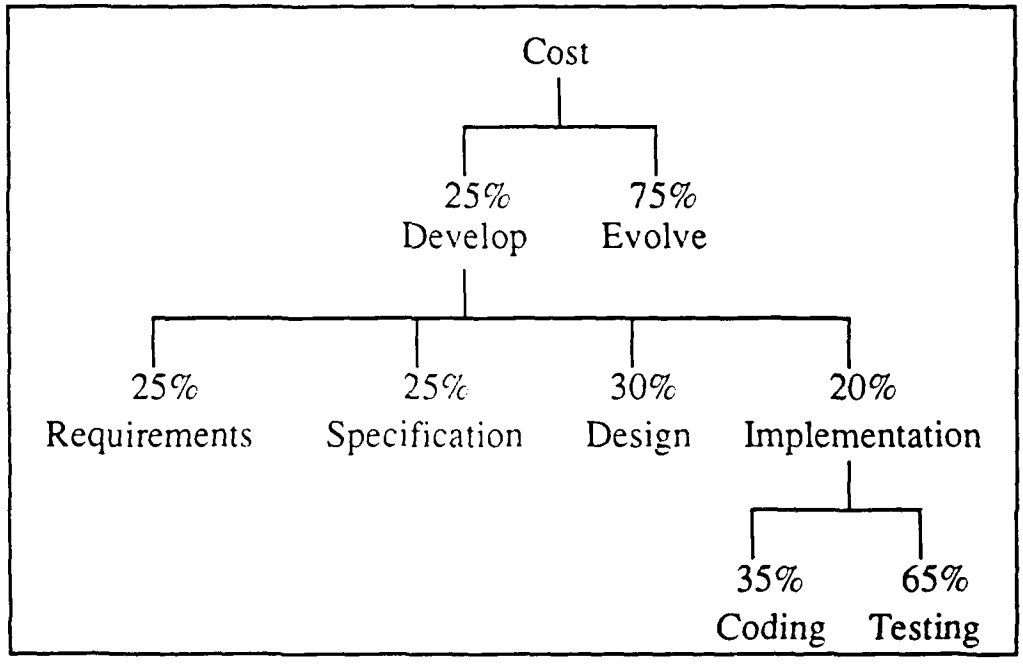


Figure 2 Cost Distribution of Software

Rapid prototyping is the construction activity leading to the prototype. The goal of rapid prototyping is to develop an executable

model of the intend system early in the development process. It is the process of quickly building and evaluating a series of prototypes.

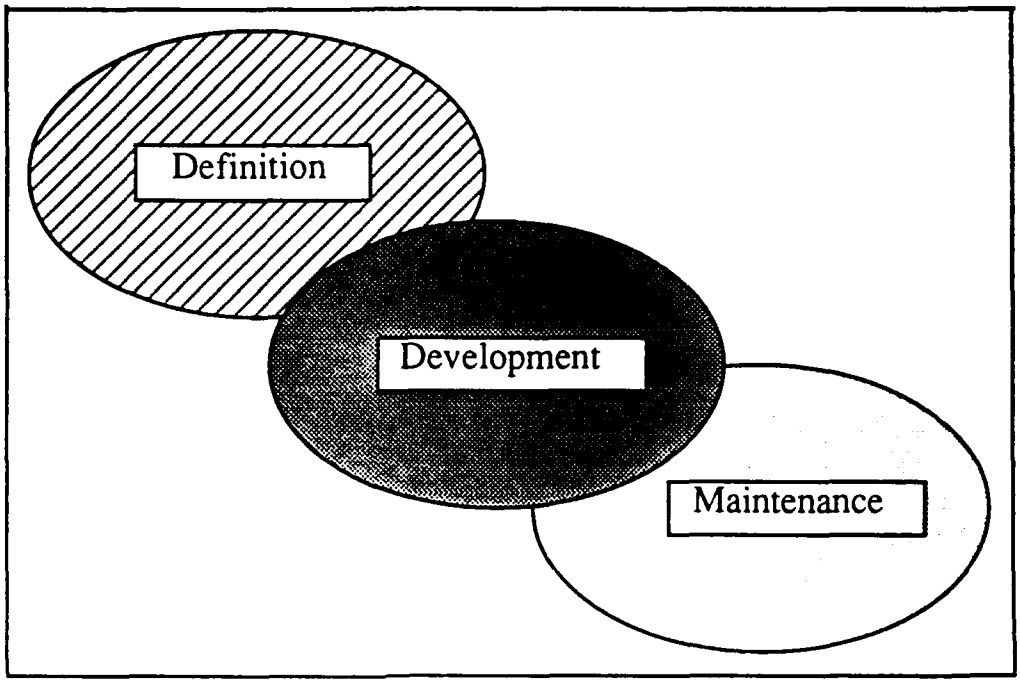
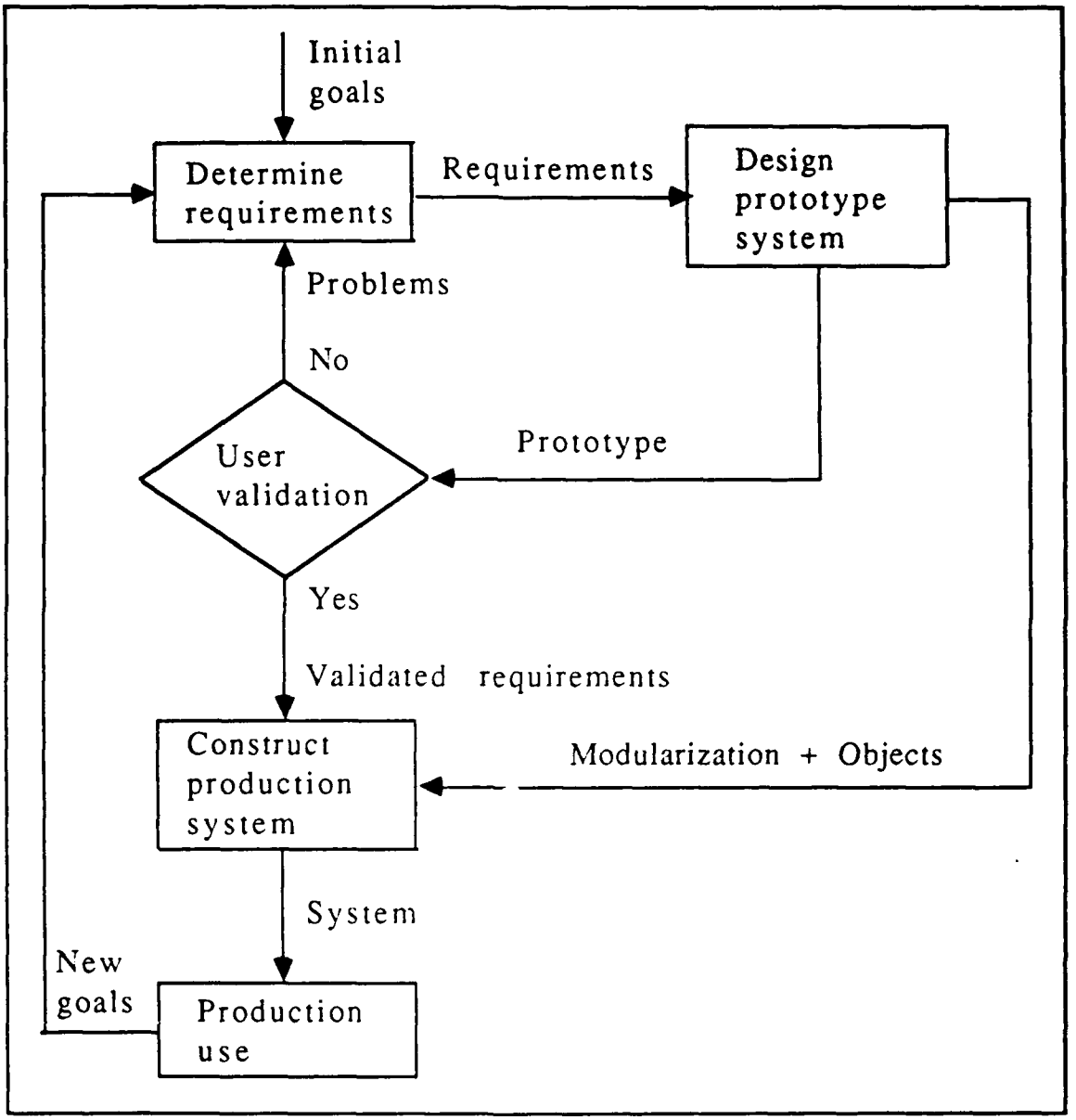


Figure 3 A Generized View of Life Cycle

Figure 4 on page 7 illustrates the iterating prototyping cycle [Ref. 27: P. 14]. The user and the designer work together to define the requirements and specifications. After requirements come out, the designer constructs a model or prototype of the proposed system in a prototype description language at the specification level. The resulting prototype is a partial representation of the system, including only those attributes necessary for meeting the requirements. It serves as an aid in analysis and design rather than

as production software. The principle of information hiding is particularly important in this context to provide flexibility.



Fugure 4 The Prototyping Cycle

The resulting prototype is validated by the user. If the prototype fails to execute properly as user expected, the user identifies problems and works with the designer to redefine the requirements. This is an interactive process. It continues until the user determines the critical aspects of the envisioned system.

The designer uses the validated requirements as well as hidden components which are created in the design of prototype to construct production system. If goals have been changed during the use of production, these new goals will trigger further iterations of the prototyping cycle.

C. CAPS AND PSDL OVERVIEW

1. CAPS

Rapid prototyping of embedded systems can be accomplished by using the Computer Aided Prototyping System (CAPS) and its associated language, Prototype System Description Language (PSDL) [Ref. 27]. The Computer Aided Prototyping System (CAPS) process is one proposed method for speeding up the design and implementation of large software systems while increasing the reliability of the final product and, at the same time, reducing the need for expensive design changes during the latter stages of software development [Ref. 33: P. 1]. This process utilizes the approach of rapid prototyping combined with a reusable software management base to produce a prototype of the system being designed. PSDL is a high level prototyping language. It supports a modeling strategy based on data

flow graphs augmented with non-procedural timing and control constraints [Ref. 23].

Development of an executable prototype with CAPS requires a modular design which supports retrieving appropriate reusable software modules. Figure 5 on page 12 illustrates the major steps that the designer uses to interact with the CAPS to develop a prototype [Ref. 19: P. 12].

The designer begins the process by entering the specifications of the intended software component. The rewrite subsystem maps the specifications into an abstract form to search for components in the software base. If the component is found and is unique, then it is retrieved. If more than one component is found, the designer has to choose the best one from them. If the component is not found, the specification can not be met by an existing component. In such a case, the designer has to decompose the specifications by using the system's prototyping language.

If the specification is not decomposable, the designer should code this basic specification in a programming language. When the specification is decomposed, new specifications are created. Since we cannot know if there are suitable components before we decompose the parent specification, these new sub-specifications can be processed in an iterative way. When a specification is decomposed into a network of simpler components, the required interconnections are recorded in the design database with a dataflow diagram, which

is part of the syntax of the prototyping language and serves as design documentation.

After the designer decomposes the specification, the entire process is applied to those specifications. The CAPS reduces the designer's efforts by automating time-consuming tasks in conventional prototyping, such as turning specifications into prototypes, modifying prototypes, and searching for available reusable components [Ref. 25].

The main subsystems of CAPS are the user interface, the software database system, and the execution support system. The user interface provides facilities for entering information about the requirements and design, presenting the results of prototype execution to the customer, guiding the choice of which aspects of the prototype to demonstrate, and helping the designer propagate the effects of a change. The user interface consists of a syntax-directed editor with graphics capabilities, and expert system for communicating with end user, a browser, and a debugger.

The software database consists of a design database, a software base, a software design-management system, and a rewrite subsystem. The design database contains the PSDL prototype descriptions for each software development project using CAPS. The software base contains PSDL descriptions and code for all available reusable software components. The software design-management system manages and retrieves the versions, refinements, and alternatives of the prototypes in the design database and the

reusable components in the software base. The rewrite subsystem translates PSDL specifications into a normalized form used by the design-management system to retrieve reusable components from the software base [Ref. 26].

The purpose of execution support system is to turn the PSDL description of the system under construction into an executable prototype using the software components that have been retrieved from the software base or written for the prototype [Ref. 33: P. 13]. The execution support system helps speed up design as well as design changes by providing a localized view of the processes in the prototype, analyzing the prototype's timing properties, and providing the ability to quickly demonstrate the consequences of design decisions through prototype execution. The execution support system contains a translator, a static scheduler, and a dynamic scheduler. During this process, validation of the critical timing information provided by the designer is done, control constraints are translated into the base language of the system, and the base language modules are organized for final execution. Figure 6 on page 15 illustrates the architecture of the execution support system.

2. PSDL

A good language for expressing design thoughts in terms of a precise model is important for rapid prototyping [Ref. 25: P. 68]. In order to produce a reasonable prototype, PSDL should meet the following subgoals:

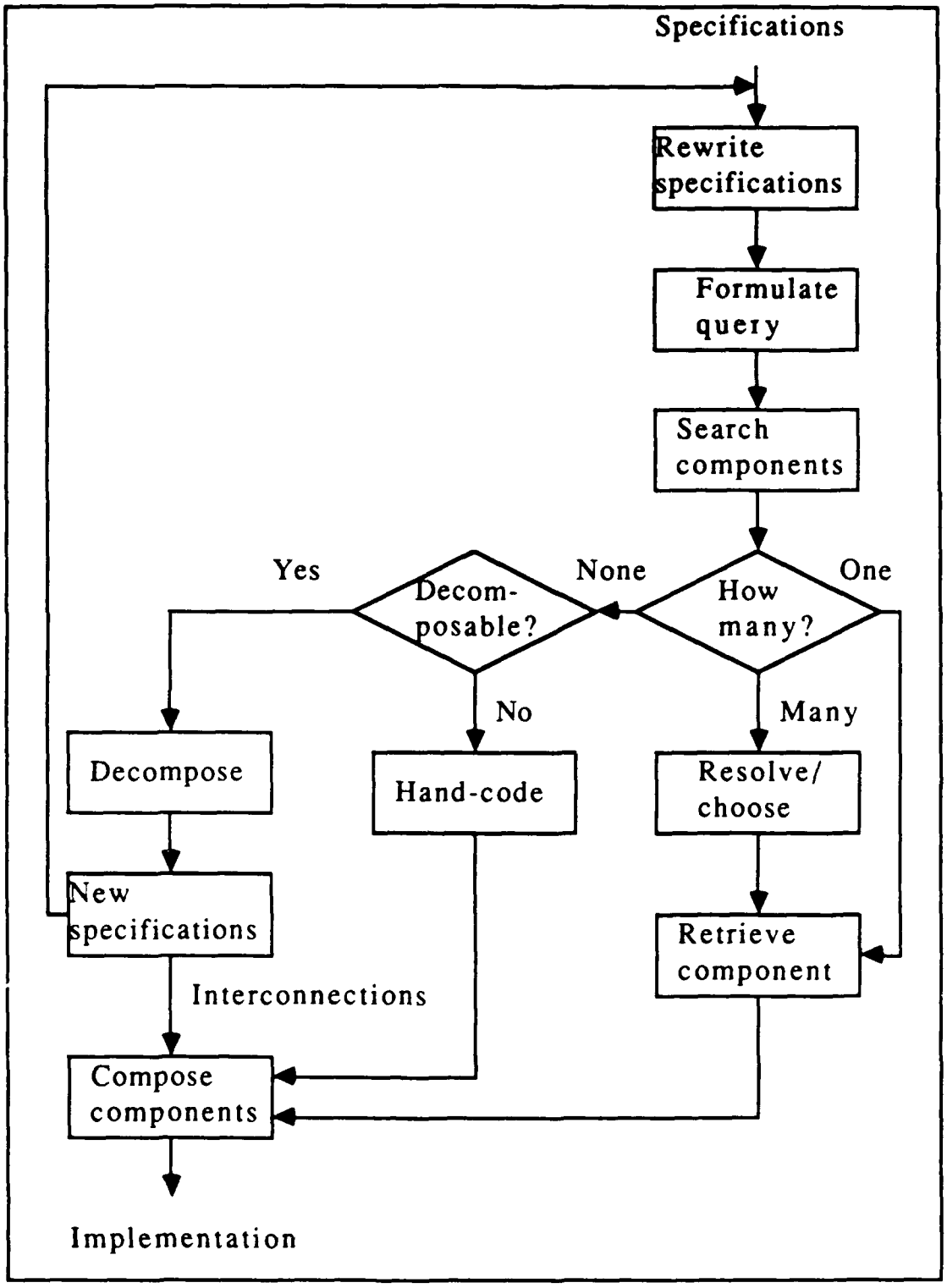


Figure 5 Prototype Development Using the CAPS

1. PSDL should be *simple* and easy to use.
2. PSDL should support good *modularity*.
3. PSDL should support retrieval of *reusable* modules.
4. PSDL should support *adaptability* for modification.
5. PSDL should support *abstraction* to the software system.
6. PSDL should support *requirements tracing*.
7. PSDL should support a *hierarchically structured* prototype.
8. PSDL should create an executable prototype.
9. PSDL should provide *graphical* notation.
10. PSDL should be well suited for use with Ada.
11. PSDL should be based on a simple computational model.

PSDL serves as an executable prototyping language at a specification or design level. It was designed as the primary connection between the designer and the components of CAPS. PSDL supports operator, data, and control abstractions, and encourages hierarchical decompositions based on both data flow and control flow.

a. PSDL Computational Model

The computational model is an augmented graph

$$G = (V, E, TC(v), C(v))$$

where V is the set of vertices, E is the set of edges, TC(v) is the maximum execution time for each vertex v, and C(v) is the set

of control constraints for each vertex v. Each vertex is an operator and each edge is a data stream.

(1) *Operators.* PSDL operators represent either functions or state machines. Simply stated, a function is a immutable module which is influenced only by the most recent stimulus and does not exhibit internal memory; while a state machine is a mutable module with an internal state. A module is *mutable* if the response of the module to at least one message can be affected by previous messages it has received, and is *immutable* otherwise.

When an operator fires, it reads one data from each input stream and writes, at most, one computed data value onto each output stream. There is a precedence relationship between each operator described as follows :

if the output from operator A is the input to operator B,

then operator A must fire before operator B.

(2) *Data Streams.* PSDL data streams are communication links between two PSDL operators, the producer (output) and the consumer (input). Each data stream is an arrow in the PSDL computational model. The precedence relationship between the data values in each data stream is described as follows :

if data value **a** is generated before data value **b**,

then **a** must be delivered to the next operator before **b**.

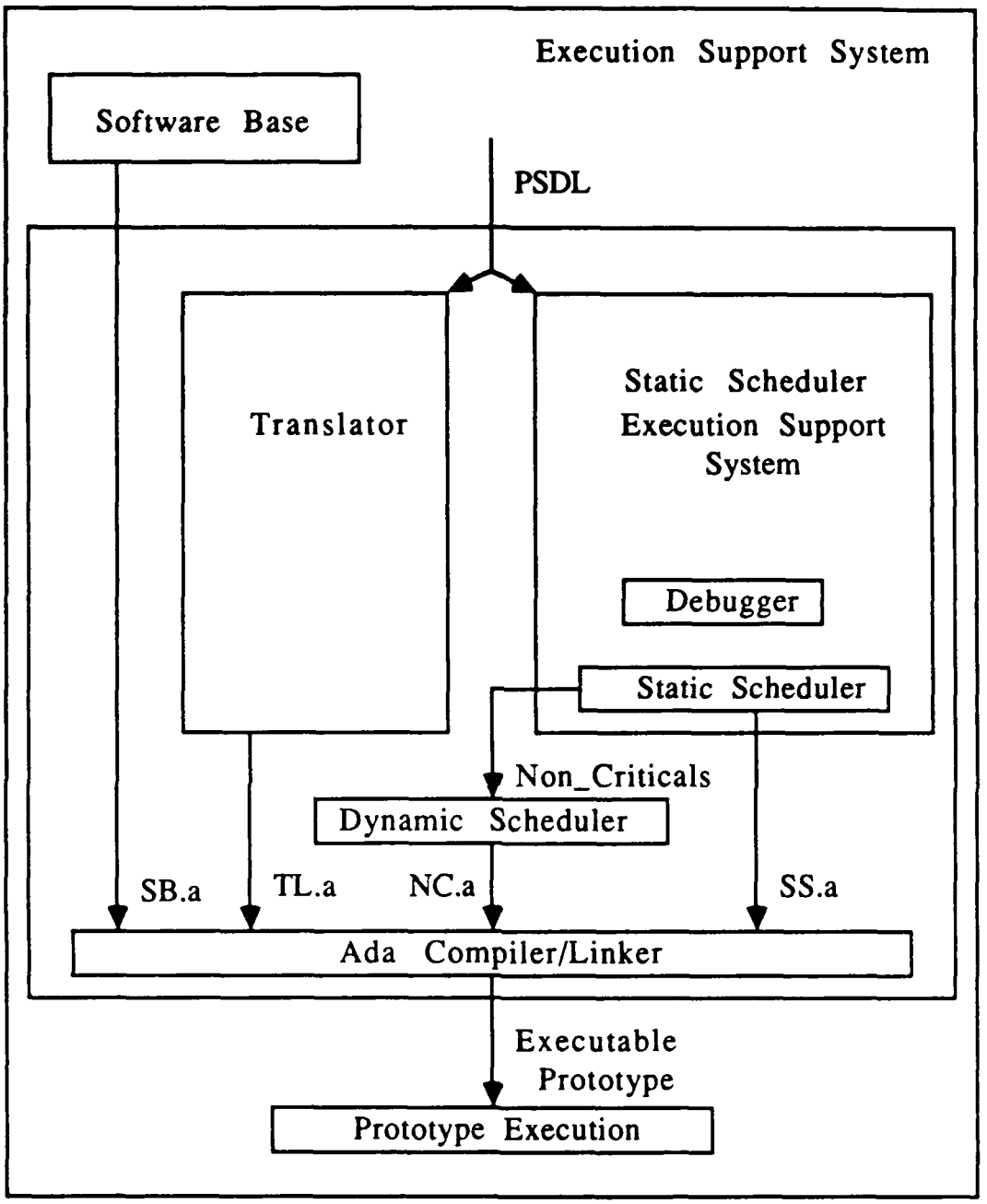


Figure 6 Execution Support System

These data streams are designed as either data flow or sampled streams. In data flow, data values are guaranteed to be not lost or repeated by utilizing a first-in first-out queue. Strict

lost or repeated by utilizing a first-in first-out queue. Strict sequencing relationships should be enforced between the producer and consumer of a data flow stream to insure that the queue does not overflow or underflow.

A sampled stream guarantees that data values can be entered into or delivered from a data stream as they are required by the operators. A sampled stream does not require or provide protection against lost or repeated values since only the most recent value is of interest. Therefore it does not require a strict sequencing relationship between the operators. Figure 7 on page 16 illustrates the graphic model of PSDL.

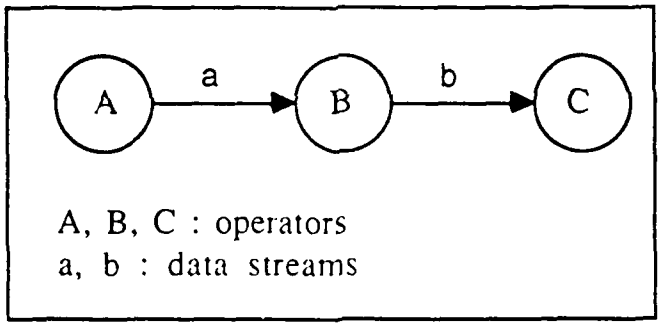


Figure 7 Graphic Model of PSDL

(3) ***Timing and Control Constraints.*** The timing and control constraints for operator firing and execution are critical in hard real-time systems. Within the computational model, each time critical operator includes a maximum execution time which gives the worst case time to complete execution once the operator fires. Critical operators can also include conditional control constraints that act as

guards. These guards stipulate firing conditions for an operator, conditions necessary before computed values are output onto data streams, or exception situations.

b. PSDL Abstractions.

Three types of abstractions represent the major building blocks for constructing the PSDL prototype. They are operator abstractions, data type abstractions, and control abstraction.

(1) *Operator Abstractions.* An operator abstraction is either a functional abstraction or a state machine abstraction. PSDL operators have two major parts: the *specification* and the *implementation*. The specification part contains attributes describing the form of the interface, the timing characteristics, and both formal and informal descriptions of the observable behavior of the operator. A PSDL operator corresponds to a state machine abstraction if its specification part contains a STATES declaration, and otherwise it corresponds to a functional abstraction.

The implementation part determines whether the operator is atomic or composite. Atomic operators have a keyword specifying the underlying programming language (Ada in our application) and the name of the retrieved reusable module that implements this operator. Composite operators have attributes which include communication graph, internal data, control constraints, and informal description.

(2) *Data type Abstractions.* Data abstractions decouple the behavior of a data type from its representation. This is

especially important in prototyping because the behavior of the intended system is only partially realized, capturing only those aspects important for the purposes of the prototype [Ref. 23: P. 1413]. The PSDL prototype language enforces explicit interactions between modules by requiring that both pre-defined and user-defined data types be immutable [Ref. 19: p. 16].

(3) *Control Abstractions.* The control abstractions of PSDL are represented as enhanced data flow diagrams augmented by a set of control constraints. Order of execution is only partially specified, and is determined from the data flow relations given in the enhanced data flow diagrams, based on the rule that an operator consuming a data value must not start until after the operator producing the data value has completed.

D. ANALYSIS OF ALGORITHMS

1. Introduction

The purpose of algorithm analysis is to predict the behavior, especially the running time, of an algorithm without implementing it on a specific computer [Ref. 32: P. 37]. Consider the following Ada-like instructions:

```
for i in 1 .. n loop
    for j in 1 .. n loop
        do_something;
    end loop;
end loop;
```

It is obvious that the instruction do_something is executed n*n times; while it is executed 2*n times when in the following codes:

```
for i in 1 .. n loop
   do_something;
end loop;
for j in 1 .. n loop
    do_something;
end loop;
```

These frequencies n and n^2 are different increasing orders of magnitude. The order of magnitude of a statement refers to its frequency of execution. The order of magnitude of an algorithm refers to the sum of the frequencies of all of its statements. Given two algorithms for solving the same problem whose orders of magnitude are n and n^2, we will prefer the first. For example, if n = 10 then these algorithm will require 10 and 100 units of time to execute respectively.

It is usually hard to predict the exact behavior of an algorithm. There are too many influencing factors. Instead, we will extract the main characteristics of the algorithm. We ignore constant factors and concentrate on the behavior of the algorithm as the size of the input goes to infinity [Ref. 32: PP. 37-38]. For example, if the number of steps is $2n^2 + 50$, then we ignore the constants 2 and 50 and say that the running time is approximately n^2. The analysis is thus only an approximation.

2. Complexity and O-, Ω-Notation

When we discuss the most "efficient" algorithm for solving a problem, the notion of efficiency involves all the various computing resources needed for executing an algorithm. However, by the "most efficient" we normally mean the fastest. Since time requirements are often a dominant factor determining whether or not a particular algorithm is efficient enough to be useful in practice, we shall concentrate on this single resource. The time complexity function for an algorithm expresses its time requirements by giving, for each possible input length, the largest amount of time needed by the algorithm to solve a problem instance of that size [Ref. 14: P. 5].

There are several kinds of mathematical notation which are very useful for the analysis of algorithm. We use two kinds of notation: O-notation and Ω-notation [Ref. 17: PP. 27-31].

Definition: $f(n) = O(g(n))$ iff there exist two positive constants c and n0 such that $|f(n)| <= c|g(n)|$ for all $n >= n0$; $f(n) = \Omega(g(n))$ iff there exist positive constants c and n0 such that for all $n > n0$, $|f(n)| >= c|g(n)|$.

If an algorithm has computing time $O(g(n))$, we mean that if the algorithm is run on some computer on the same type of data but for increasing values of n, the resulting times will always be less than some constant times $|g(n)|$. When determining the order of magnitude of f(n) we shall try to obtain the smallest g(n) such that $f(n) = O(g(n))$.

O-notation is used to express an upper bound for the execution time of an algorithm while Ω-notation is to determine a lower bound.

A polynomial time algorithm is defined to be one whose time complexity function is O(p(n)) for some polynomial function p, where n is used to denote the input length. Examples of algorithms whose time complexity function cannot be so bounded are exponential time algorithms. Table 1 on page 22 illustrates the comparison of several polynomial and exponential time complexity functions [Ref. 14: P. 7]. The distinction between polynomial time algorithms and exponential time algorithm admits of many exceptions when the problem instances of interest have limited size. Even in Table 1, the 2^n algorithm is faster than n^5 algorithm for $n <= 20$.

When the size n is small, the difference between polynomial or exponential time algorithms does not matter. However, as n gets large, there are large differences between them. This table indicates some of the reasons why polynomial time algorithms are generally regarded as being much more desirable than exponential time algorithms.

E. NP PROBLEMS

1. Introduction

There is wide agreement that a problem has not been "well-solved" until a polynomial time algorithm is known for it. A problem is said to be *intractable* if it is so hard that no polynomial time algorithm can possibly solve it [Ref. 14: P. 8].

Time Complexity function	Size n					
	10	20	30	40	50	60
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second
n^3	.001 second	.008 second	.027 second	.064 second	.125 second	.216 second
n^5	.1 second	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13.0 minutes
2^n	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	.059 second	58 minutes	6.5 years	3855 centuries	$2*10^8$ centuries	$1.3*10^{13}$ centuries

Table 1 Comparison of several polynomial and exponential time complexity functions

There are many problems for which no polynomial-time algorithm is known. Some of these problems may be solved by efficient algorithms that are yet to be discovered. However, it is very possible that there exist many problems which can not be solved efficiently.

2. Deterministic Turing Machines and the Class P

In order to formalize the notion of an algorithm, we need to fix a particular computational model. There are many computational models; however, the Turing machine model is most widely used.

The Turing machine consists of a control mechanism that can be in one of a finite number of states at any given time. One of these states is called the initial state and represents the state in which the machine starts a computation. Another state is the machine's halt state. Whenever the halt state is reached, the Turing machine stops. A Turing machine can both read from and write on its input medium. It is equipped with a tape head that can be used both to read and write symbols on the machine's tape.

If M is a Turing machine, we say that M accepts the language L in polynomial-time if the machine M accepts all input strings in L, rejects all input strings not in L, and there is a polynomial p(n) such that the number of steps required to accept any $w \in L(M)$ is no greater than p(|w|). We define P to be the class of languages that can be accepted by Turing machine in polynomial-time [Ref. 4: P. 260]. In our application, we regard the problem as an input language L for the test of Turing machine. We will use the words "language" and "problem" alternately.

3. Nondeterministic Computation and the Class NP

A nondeterministic Turing machine is similar to a traditional Turing machine. The distinction is that a nondeterministic machine may provide more than one applicable transition for some current state/symbol pair. If a nondeterministic Turing machine should arrive at a current state/symbol pair from which more than one transition is applicable, the machine makes a nondeterministic choice

and proceeds with the computation by executing one of the applicable options.

We say that a nondeterministic Turing machine M accepts the language L in polynomial-time provided L = L(M) and there is a polynomial p(x) such that for any w ∈ L, M can accept w by a computation involving no more than p(|w|) steps. Furthermore, we define NP to be the class of languages that can be accepted by nondeterministic Turing machines in polynomial-time [Ref. 4: P. 270].

4. The Relationship Between P and NP

Since every deterministic Turing machine is contained in the class of nondeterministic Turing machines, we can immediately claim that P ⊆ NP. But the question of whether P = NP is not yet resolved. There are many decision problems throughout computer science that can be restated in terms of recognizing languages that are known to be in NP but whose membership, or lack of membership, in P is not yet determined. A decision problem is a problem that can be stated in the form of a question whose answer is either yes or no. Thus, if P = NP, these problems would appear to have practical algorithmic solutions, but if p ≠ NP, the chances of finding efficient algorithm to these problems would be significantly reduced [Ref. 4: PP. 270-271].

There are many problems belong to NP but no polynomial time solution algorithms have been found despite the efforts of many knowledgeable and persistent researchers. There is a widespread belief that P ≠ NP. Figure 8 on page 25 shows the world of NP

problem [Ref. 14: p. 34]. We expect that the shaded region denoting NP-P is not totally uninhabited.

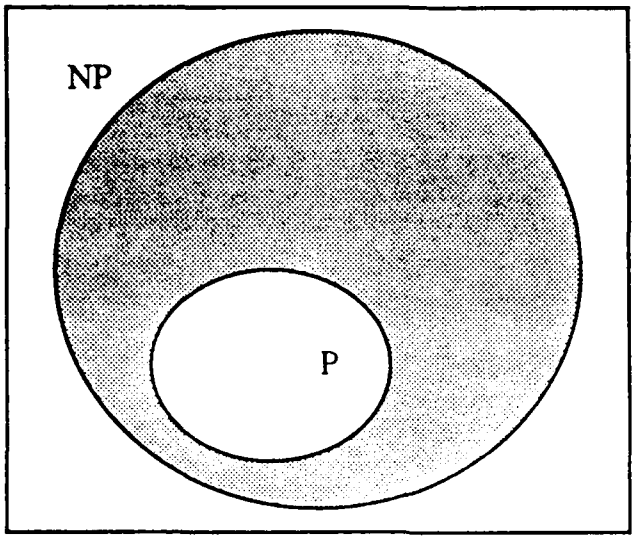


Figure 8 The World of NP

5. Polynomial Transformations and NP-Completeness

If P differs from NP, then all problems in P can be solved with polynomial-time algorithms while all problems in NP-P are intractable. Thus, given a decision problem $\Pi \in$ NP and Figure 8, we would like to know which of these two possibilities holds for Π.

Unless we can prove that P $\neq$ NP, there is no possibility to show that any problem belongs to NP-P. The theory of NP-completeness does not provide a method to prove that P $\neq$ NP. Nor does it provide a method of obtaining polynomial time algorithms for the problems belong to NP. Instead, theory shows that "if P $\neq$ NP, then $\Pi \in$ NP-P." If a problem is NP-complete, it will have the

property that it can be solved in polynomial time iff all other NP-complete problems can also be solved in polynomial time.

Let L_1 L_2 be problems, L_1 transforms to L_2 (written as $L_1 \propto L_2$) if and only if there is a way to solve L_1 by a deterministic polynomial-time algorithm using a deterministic algorithm that solves L_2 in polynomial time. There are two lemmas for the transformation [Ref. 14: PP. 34-37].

Lemma 1: If $L_1 \propto L_2$, then $L_2 \in P$ implies $L_1 \in P$.

Lemma 2: If $L_1 \propto L_2$ and $L_2 \propto L_3$, then $L_1 \propto L_3$.

Formally, a language is defined to be NP-complete if $L \in NP$ and, for other languages $L' \in NP$, $L' \propto L$. Lemma 1 leads us to our identification of the NP-complete problems as "the hardest problems in NP." Lemma 2 tell us that if any NP-complete problem can be solved in polynomial time, then all problems in NP can be so solved. On the other hand, if any problem in NP is intractable, then so are all NP-complete problems. Figure 9 on page 27 illustrates the scope of NP-complete problems [Ref. 14: P. 37].

6. Dealing with NP-Complete Problems

Many scheduling problems have been shown to be NP-complete. Table 2 on page 28 illustrates the survey [Ref. 6: P. 20]. The scheduling problems stated in Table 2 assume that there are to timing constraints for tasks. The algorithms solving problem 1 and problem 2 are optimal if their time complexities are not greater than their corresponding problem complexities.

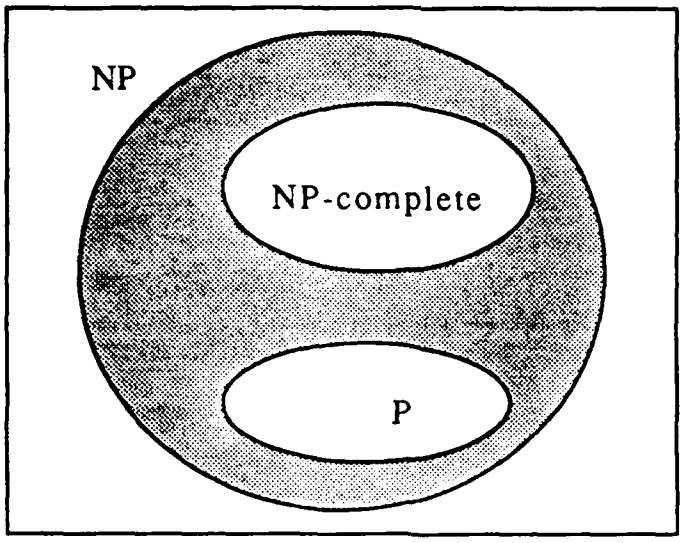


Figure 9 The World of NP, Revisited

The optimal solutions for the NP-complete problems refer to least time to schedule all tasks and/or least processors to schedule them. If we can prove that every step in algorithm guarantees these properties, we say this algorithm optimal.

The notion of NP-completeness is a basis that allows us to identify problems for which no polynomial algorithm is likely to exist. But we still need to solve such problems. In doing so, we have to sacrifice something to comprise. The most compromises concern the optimality, robustness, guaranteed efficiency, or completeness of the solution [Ref. 32: P. 357].

The best known algorithms for NP-complete problems have a worst case complexity that is exponential in the number of inputs. We can try to solve NP-complete problems in polynomial time on the average. However, finding the right distribution is usually very

difficult. Another possibility is algorithms whose running times are exponential, but they work reasonably well for small input, which may be sufficient.

Problem Number	Number of processors m	Task processing time t_j	Precedence constraints	Complexity
1	Arbitrary	Equal	Forest	O(n)
2	2	Equal	Arbitrary	$O(n^2)$
3	Arbitrary	Equal	Arbitrary	NP-complete
4	Fixed m>=2	t_j = 1 or 2 for all i	Arbitrary	NP-complete
5	Arbitrary	Arbitrary	Arbitrary	NP-complete

Table 2 Complexity of Nonpreemptive Scheduling Problems

A feasible solution with value close to the value of an optimal solution is called an approximate solution. An approximate solution may not lead to the precise result. However, there are many problems that have no exact solution, we have to use approximate methods to solve them. We need a heuristic to get approximations to the optimal solution.

In PSDL, a feasible solution is a sequence of tasks which meet not only precedence but also timing constraints. If one or more tasks in the sequence violate these two constraints, this sequence cannot

be feasible. If we cannot find any feasible solution in a scheduling system, we have to find an optimal solution.

An optimal solution in PSDL is one of the sequences which have no feasible solutions. In addition, this solution is the best sequence among them. The word "best" may refer to least tardiness time for the sequence or least number of processors needed to executed the schedule.

F. OBJECTIVES

This thesis describes the design for the *static scheduler* system (see Figure 6 on page 15). The objective of this thesis is to present the algorithms which successfully schedule the tasks in multiprocessor systems with consideration of the precedence constraints on such tasks.

G. ORGANIZATION

Chapter II provides a survey of the static scheduling algorithms in hard real-time systems. Chapter III designs the optimal scheduling algorithms for handling graph-based hard real-time specification. Chapter IV contains the conclusions and recommendations for the future work.

II. SURVEY OF PREVIOUS WORK

Much research has been done on hard real-time scheduling problems. There are different kinds of problems corresponding to scheduling. We describe the scopes of scheduling problems in this chapter as well as the current research related to each.

A. SOME DEFINITIONS AND TERMS ABOUT SCHEDULING PROBLEMS

We introduce some concepts and definitions which are often used in scheduling problems.

1. PROBLEM CLASSIFICATION

Scheduling approaches and scheduling systems are classified as static or dynamic, and as centralized or distributed.

a. Static approaches and dynamic approaches

Algorithms for scheduling tasks in hard real-time systems can be classified as *static* or *dynamic*. A static approach calculates schedules for tasks off-line and requires complete prior knowledge of tasks' execution times. A dynamic approach determines schedules for tasks on the fly and allows tasks to be dynamically invoked. The advantage static approaches is their low run-time cost; however, they are inflexible and cannot adapt to a changing environment or to an environment whose behavior is not completely predictable. The dynamic approaches are flexible and can easily adapt to changes in the environment, but they involve higher run-time cost. [Ref. 39: PP. 150-173]

b. Centralized system and distributed system

A centralized system is one in which the processors are located at a single point in the system and the inter-processor communication cost is negligible compared to the processor execution cost. A multiprocessor system with shared memory or a multiterminal system using the same processor is the example of such system. In contrast, a distributed system is one in which the processors are distributed at different points in the system and the inter-processor communication cost is not negligible compared to the processor execution cost. A local area computer network is an example of such system.

2. PROBLEM DEFINITION

A scheduling problem in a hard real-time system is defined by the model of the system, the nature of tasks to be scheduled, and the objectives of a scheduling algorithm.

a. System models

A hard real-time system consists of two parts: hardware and software. Hardware includes those devices which are required to execute all the strategies in hard real-time system. Those which not belong to hardware are softwares. Software includes PSDL, static and dynamic schedulers etc. Current PSDL does not represent hardware structures. The resources refer to those things which are available for the scheduling. They might be hardware devices, data, or programs; however, the only resources shared by tasks in PSDL are data streams and processors.

b. Nature of tasks in PSDL

A task is a software module that can be invoked to perform a particular function. A task is the scheduling entity in a system [Ref. 8]. A task corresponds to a PSDL operator and represented as a vertex in the PSDL implementation graph. A task is characterized by its timing constraints, precedence constraints, and required processors.

Tasks in hard real-time systems can be distinguished as preemptive and nonpreemptive. A task is preemptive if its execution can be interrupted by other tasks and resumed at the point of suspension. A task can still be preemptive if the times when the task can be interrupted are constrained. A task is nonpreemptive if it must run to completion once it starts. [Ref. 6] compares preemptive and nonpreemptive task schedules. Preemption can make tasks easier to schedule. For example, consider a task system consisting of three independent tasks of length 2 to be scheduled on two processors. A nonpreemptive schedule for the system takes 4 units of time. A shortest preemptive schedule on the other hand takes 3 units; a savings of 25%. Figure 10 on page 33 illustrates the comparison [Ref. 6: p. 51]. However, the preemptive schedule also have the following disadvantages:

(1). Preemption has overhead for context switching time which is illustrated on Figure 10(c). If the schedule is long or the switching is frequent, it could not meet the Real-time requirements.

(2). Preemption is difficult to implement in Ada.

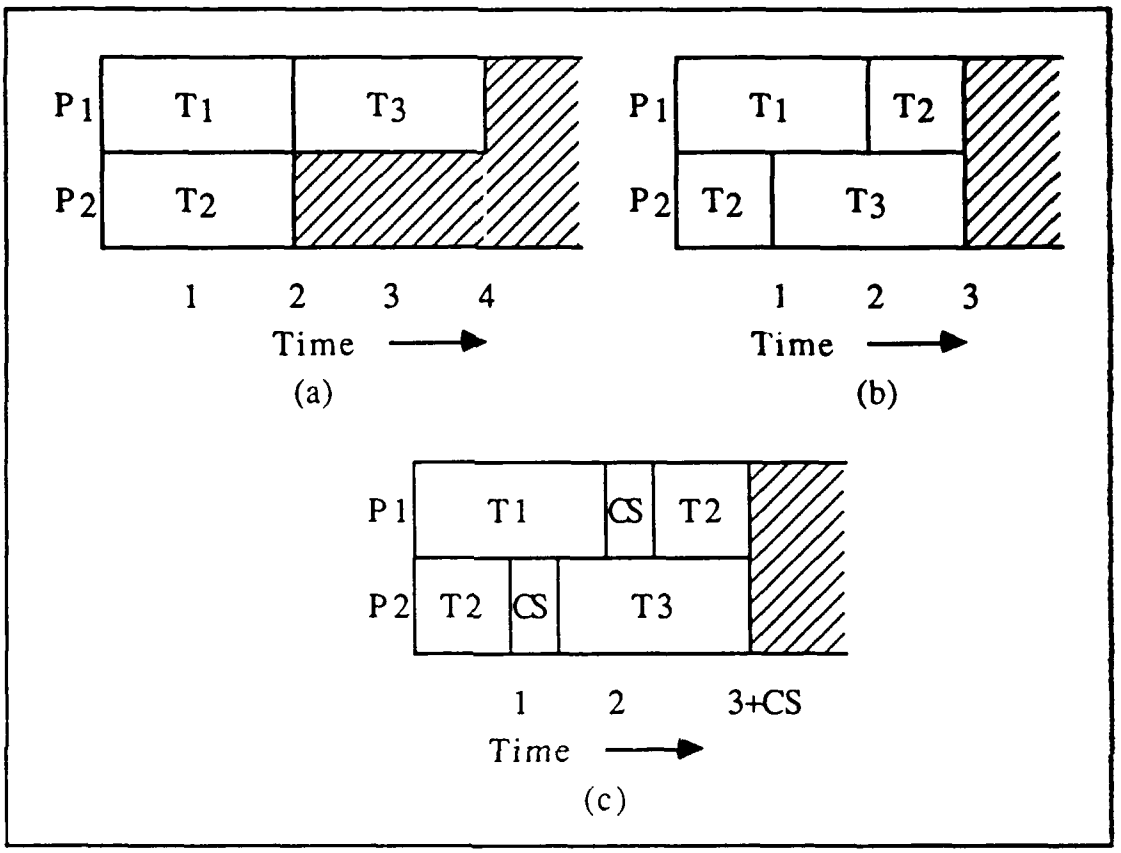


Figure 10 The Comparison of Preemptive and Nonpreemptive Schedules.

(a) Nonpreemptive (b) Preemptive schedules for a system of three independent tasks of length 2 scheduled on two identical processors and (c) Preemptive schedules accounting for context switching

Each task can be either periodic or nonperiodic. A periodic task is defined as one which is invoked exactly once per period P. Within P, the task must be scheduled to execute. Every periodic task must have a specified period and may have a specified

deadline. A nonperiodic task is one which is expected to execute just once each time it is invoked. In PSDL tasks are invoked by the arrival of new data values. It has an unknown arrival time and may have a known deadline.

(1) ***Timing Constraints.*** The timing constraints of a task are specified in terms of one or more of the following parameters:

1. The arrival time, A: The time at which a task is invoked in the system.

2. The ready time, R: The earliest time at which a task can begin execution. The ready time of a task is equal to or greater than its arrival time. When task is invoked but no processors are available, the ready time is greater than the arrival time and the task has been idle. The ready time refers to start time and the arrival time refers to earliest start time in PSDL.

3. The maximum execution time, MET: The maximum length of the execution interval (EI) for the task. The execution interval is the time length which a processor needs to execute the task. The MET represents CPU time rather than real-time.

4. The deadline, D: The time within which a task must finish. The deadline of each periodic task is an offset from its initial instance and is represented as FINISH_WITHIN in PSDL.

5. The maximum response time, MRT: The upper bound on the response time of a nonperiodic task (optional). The response time is measured from the end of the execution interval for the

producer operator of the triggering data value to the end of the execution interval for the consumer operator of the triggering data value. It is the deadline for nonperiodic tasks.

6. The minimum calling period, MCP: The lower bound on the calling period of the nonperiodic task (optional). The calling period of an operator is the length of time between the end of the execution interval for the producer of the triggering data value and the end of the execution interval for the producer of the next triggering data value. The calling period of nonperiodic task can vary from one invocation to the next, unlike the period of periodic task, which is fixed throughout the scheduling process. The calling period is shown on Figure 12 on page 37.

A task is *time-critical* if it has at least one timing constraint associated with it, and is *non time-critical* otherwise. The starting time plus the MET must not greater than deadline. The degree of freedom enjoyed by the static scheduler is characterized by the slack, which is defined as the difference of deadline and MET.

Each periodic task must have a period and may have a deadline. These two timing constraints partially determine the set of scheduling intervals (SI) for the task. The scheduling interval is an interval of time within which a task have to be completed and the deadline must be met. Each periodic task must be fired exactly once in each scheduling interval, and must complete execution before the end of the scheduling interval. The period is the length of time between the start of any scheduling interval and the start of the next

scheduling interval. Figure 11 on page 36 illustrates the relation between the timing constraints, scheduling intervals, and execution intervals for a periodic task [Ref. 29: P. 10]. For the case of multiprocessor schedules, it sometimes makes sense for the length of the scheduling interval to be longer than the period. In such cases adjacent scheduling intervals may overlap.

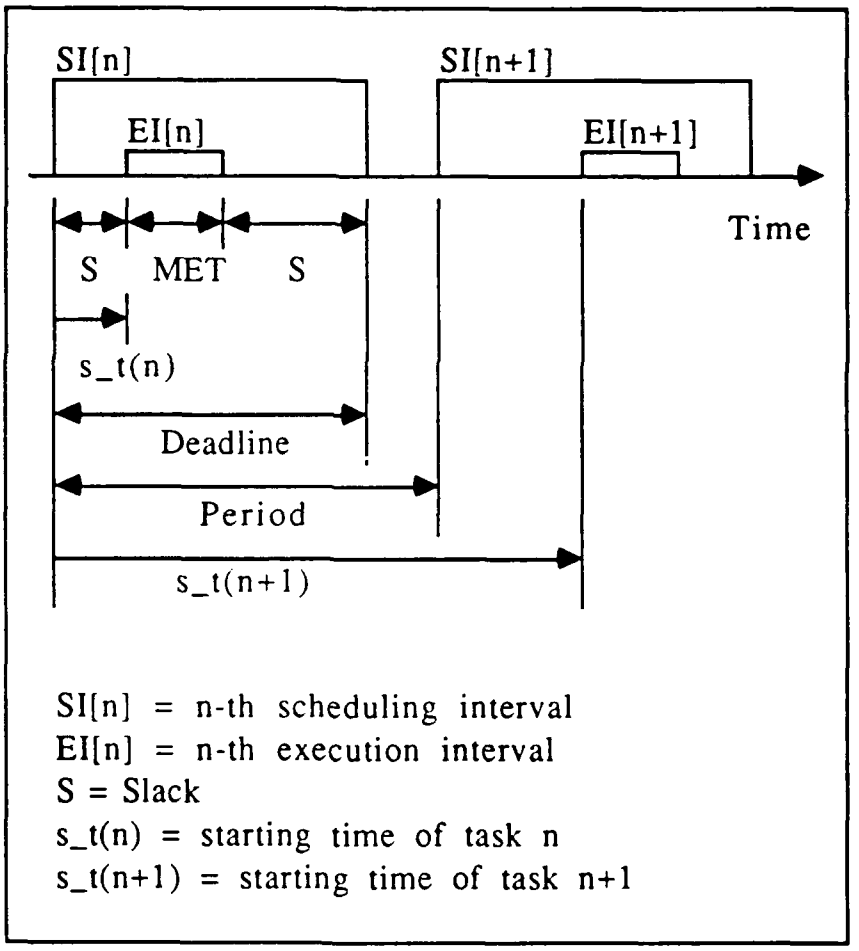


Figure 11 Timing Constraints for a Periodic Task

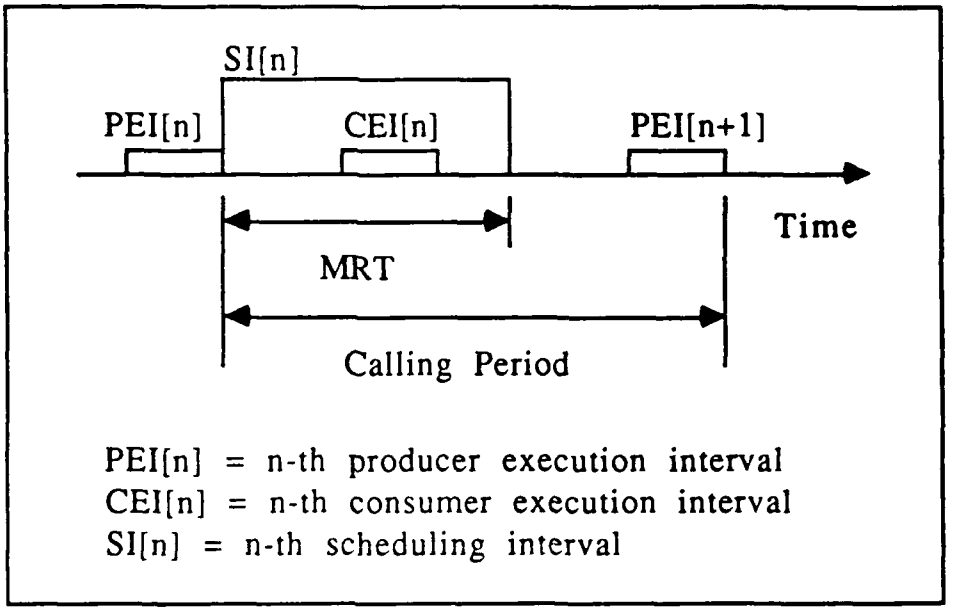


Figure 12 Timing Constraints for a Nonperiodic Task

Timing constraints for nonperiodic tasks are optional. Nonperiodic tasks with timing constraints must have both a maximum response time and a minimum calling period in addition to an MET. Figure 12 on page 37 illustrates the timing constraints for a nonperiodic task [Ref. 29: P. 12].

(2) ***Precedence Constraints.*** The task system in hard real-time scheduling is defined as $(T, \prec)$, where T is a set of tasks to be scheduled on m processors and $\prec$ is a partial order on T that specifies precedence constraints between tasks. A task T_i is said to precede task T_j if T_i must finish before T_j begins. The precedence relationships among tasks form a graph. If two tasks T_i and T_j are processed on the same processor and $T_i \prec T_j$, we have

$$\text{starting\_time}(T_i) + \text{MET}(T_j) \leq \text{starting\_time}(T_j)$$

where < holds when the processor is idle. If T_i and T_j are processed on different processors, this expression does not hold because they occupy different resources.

(3) ***Resource Requirements.*** Resources in a computer system can be categorized into two classes: active resources and passive resources. An *active resource* has processing power and can be used by at most one task at a time; CPU's and I/O processors are instances of this type of resource. A *passive resource* typically can be used in two different modes: When in *shared mode*, several tasks can use the resource simultaneously; when in *exclusive mode*, only one task can use it at a time. A file in a computer system is an example of a passive resource: a file can be read by multiple users simultaneously but can be written only by a single user at a time [Ref. 41]. A scheduling problem with resource constraints consists of a set of n tasks with given execution times, m processors, and precedence constraint $\prec$, together with a set $R = \{R_1,...,R_s\}$ of resource constraints. Each R_j is a function which maps the tasks into the nonnegative integers, indicating the amount of the i-th resource required by the task.

c. Objectives of scheduling algorithms

The function of a scheduling algorithm is to determine, for a given set of tasks, whether a schedule (the sequence and the time periods) for executing the tasks exists such that the timing, precedence, and resource constraints of the tasks are satisfied, and to calculate such a schedule if one exists.

B. DESCRIPTION OF SOME SCHEDULING ALGORITHMS

1. THE TOPOLOGICAL SORT ALGORITHM

This scheduling algorithm is used in the initial version of CAPS [Ref. 35]. The first level data flow diagram for the static scheduler in this model is illustrated in Figure 13 on page 39 [Ref. 35: P. 23]. The five components(bubbles) in the diagram are described below.

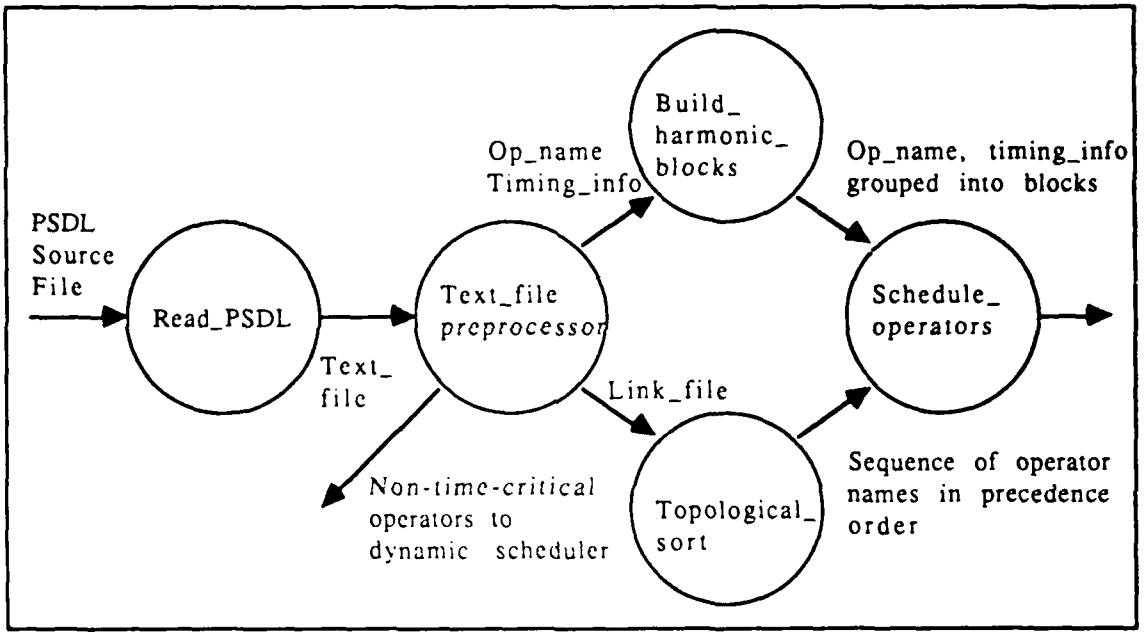


Figure 13 First Level Data Flow Diagram

In Read_PSDL, the static scheduler reads the PSDL source file and collects operator names and timing information. The resulting file is then run through a Text_file_preprocessor where operators are separated into time critical and non-time critical files. The non-time critical operators are sent to the dynamic scheduler. The dynamic

scheduler will schedule non-time critical operators into idle time slots as well as any time remaining if a time critical operator completes execution prior to its worst case execution time.

A harmonic block is a set of operators such that each operator in the set has a period that is an exact multiple of the base period and at least one of the operators has a period equal to the base period. All of the operators in a harmonic block are required to have periods that are exact multiples of the base period. Therefore, a sporadic operator must be assigned a period which is known as its equivalent period. To simply the algorithm, the operators are sorted by period in ascending order. The base period is the greatest common divisor (GCD) of the periods for all the operators in the set.

The precedence relationship among operators in the final static schedule is done in Topological_sort. It is a simple algorithm that repeatedly finds an operator which must precede all others in a set, concatenates that operator to a sequence of operators, and then deletes that operator and all its edges from the set. This cycle is repeated until all operators have been deleted from the set. The final sequence then contains all operator names in a precedence order.

Finally, the operators within each harmonic block are scheduled according to the precedence given by the topological sort and the period constraints. Figure 14 on page 41 illustrates the second level data flow diagram of the step [Ref. 35: p. 43]. The topologically sorted schedule and the harmonic block schedule are

combined at Select_next_operator. The Next_firing_interval is calculated by the following formula [Ref. 35: p. 43]:

Next_firing_interval =

[(Start time + period), (Start time + 2 * period - MET)]

The lower bound of this formula ensures that at least the length of one period will pass before the operator is scheduled to fire again. The upper bound ensures that an operator is scheduled early enough so that it can finish execution prior to the end of its period.

The theoretical development for the algorithms is available in [Ref. 35]. It contains five algorithms: Topological sort of the precedence relationship, Finding the harmonic block using GCD in both single processor and multiple processors, Finding block length, and schedule the operators. The implementation of this development and the analysis of its performance is described in [Ref. 33].

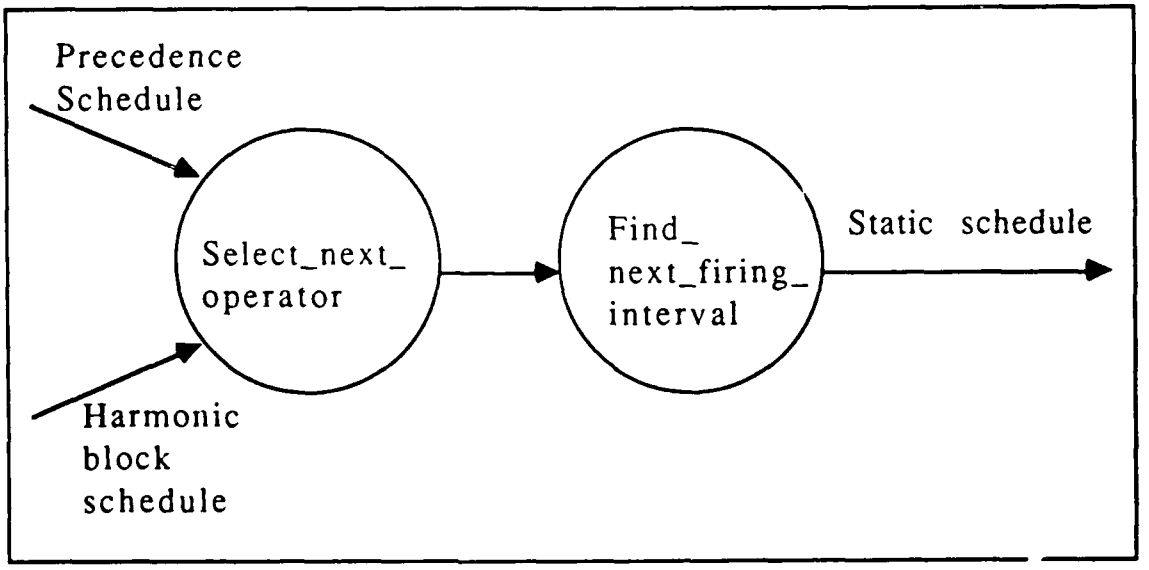


Figure 14 Schedule_operators, 2nd Level Data Flow Diagram

2. THE CRITICAL PATH METHOD

This method is proposed by Hu [Ref. 18]. The method deals with a new sequencing problem in which n jobs with ordering restrictions have to be done by men of equal ability. In our context, the n jobs represent n independent tasks and the m men of equal ability represent m identical processors.

Let N_i (i = 1, 2, ..., n) be n jobs that have to be done with technological ordering restrictions. $N_i \prec N_j$ if N_i must precede N_j, $N_i \sim N_j$ if there is no ordering restriction between the two nodes. The whole ordering restriction can be represented by a graph G consisting of n nodes representing jobs and directed arcs representing ordering restrictions. To ensure the jobs represented by the graph G are feasible, there should be no cycles formed by directed arcs of G. Figure 15 on page 43 illustrates an arbitrary graph of the model.

Although a graph G may have more than one final node, the assumption of only one final node does not lose generality. If there is more than one final node in the graph G, we can create an artificial node that is preceded by all final nodes in the graph and label it with $\alpha_i = 1$. For other nodes N_i, the label α_i is the length of the longest path from N_i to this artificial final node. Hence, we shall assume that the graph G has only one final node in the following. A node N_j is called a starting node in the graph if there does not exist a node N_i such that $N_i \prec N_j$. A node N_k is called a final node in a graph G if

there does not exist a node N_i in G such that $N_k \prec N_i$. A node N_j is called a current starting node if there is no node N_i in the current graph such that $N_i \prec N_j$. In Figure 15, N_1 is a final node and $N_{10}, N_{11}, N_{12}, N_{13}$ are starting nodes. The length of a path is the number of directed arcs in it. In Figure 15, the length of the path from N_{11} to N_1 is 3.

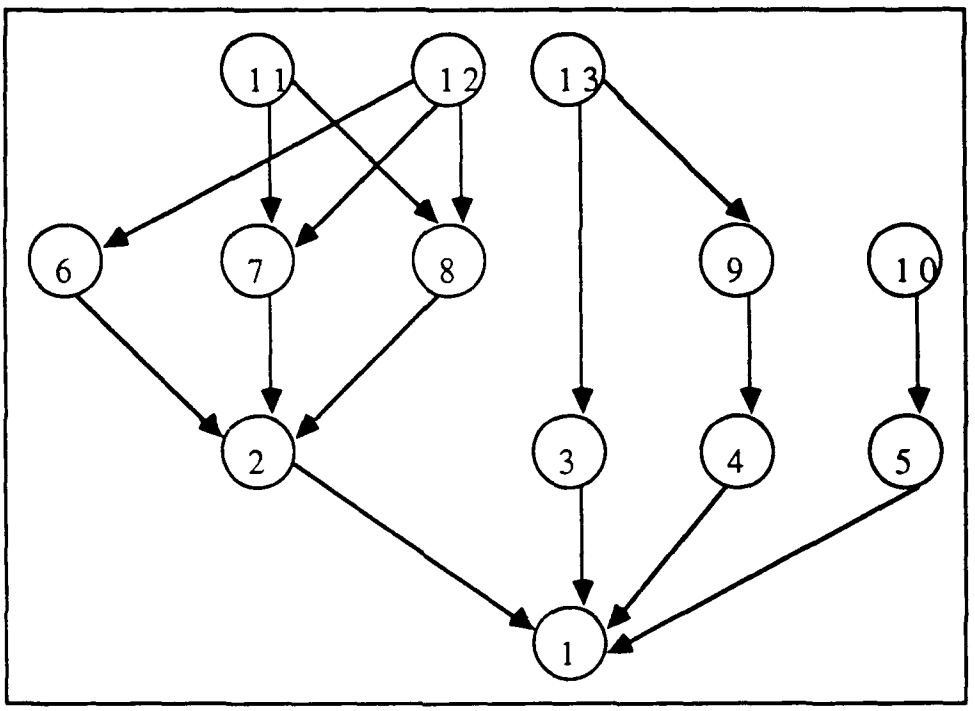


Figure 15 Example of Task Graph G

Assumptions:

1. All jobs require equal time which is one unit.

2. When a processor finishes a task, he can immediately start on another.

3. The ordering restrictions on tasks are arbitrary.

4. There are no cycles formed by directed arcs of task graph G.

Labelling process:

The labelling process assigns the path length labels α_i to the nodes N_i in the graph G in the following way.

A node N_i is labelled with $\alpha_i = x_i$ if x_i is the length of the longest path from N_i to the final node in G. The final node has the label 1. The process of labelling is equivalent to finding the longest path from a node to the final node in G.

The labelling process can be done by starting with the final node and tracing backwards. If a node can receive more than one number (branching node in the graph), label it with largest number it can receive.

Let $p(\alpha)$ be the number of nodes with label α. In Figure 15, we have $p(1) = 1$, $p(2) = 4$, $p(3) = 5$, and $p(4) = 3$. Let $s(\alpha)$ be the number of starting nodes which have label α. In Figure 15, $s(1) = 0$, $s(2) = 0$, $s(3) = 1$, and $s(4) = 3$.

Assume that we start from time $t = 0$. The subscript t is used to indicate that t units of time have passed when the current graph is obtained. Nodes that are finished are removed from the graph. As time goes on, the current graph changes. The properties $p_t(\alpha)$, $s_t(\alpha)$, and α_t are defined using the current graph at time t.

Algorithm:

1. Label all nodes with $\alpha_i = x_i + 1$ where x_i is the length of the longest path from N_i to the final node in the tree.

2. If the total number of starting nodes is less than or equal to m, where m is the number of processors available, then choose all starting nodes for execution at the current time.

3. If the total number of starting nodes is greater than m, choose m starting nodes with values of α_i greater than or equal to those not chosen. In the case of a tie, the choice is arbitrary.

4. Repeat the above steps for the remaining graph. The algorithm stops when there are no nodes in the current graph.

Validation:

The algorithm can be described as 'cutting the longest queue'. Although the algorithm is plausible and the idea is straightforward, the proof that it completes all jobs at the earliest time is somewhat long [Ref. 18: P. 846]. We will not discuss details of the proof here.

This algorithm assumes there are precedence constraints in tasks as well as identical processors. These assumptions are similar in PSDL. However, there are some deviations from PSDL.

1. PSDL assumes each task in operator has equal MET, but not unit time.

2. There could be cycles formed in PSDL data flow diagram.

3. This algorithm appeals to search the tasks graph as soon as possible, but does not test if every task meets its deadline. In fact, there are no deadlines for tasks.

3. THE CRITICAL PATH METHOD WITH UPPER BOUND AND/OR LOWER BOUND

This method optimally schedules a sequence of interrelated computational tasks on a multiprocessor computer system. The scheduler is created statically. There exists a partial ordering between the tasks. Tasks are assumed to be nonpreemptive and all tasks are assumed to require one unit of processing time.

Finding the bounds:

There are two lower bounds to be determined: the minimum number of processors required to process a task graph G in the smallest possible amount of time and the minimum time required to process a task graph G given a fixed number k of processors. There are two partitions of the task graph; namely, E and L partitions. The tasks T_i can be partitioned into ℓ subsets (E_1, E_2, ..., E_ℓ) called the earliest (or E or column) precedence partitions such that

$$\bigcup_{i=1}^{\ell} E_i = T$$

where $T = \{T_1,...T_m\}$ is the set of tasks and $E_i \cap E_j = \varnothing$ when $i \neq j$. The meaning of the E-precedence partitions is as follows. E_1 is the subset of tasks that can be initiated and executed in parallel at the very start. E_2 is the subset of tasks that can be initiated and executed in parallel after the tasks E_1 are done and so on. The elements of E_i represent those tasks that can be processed at the earliest time corresponding to level i.

The tasks {T} can also be partitioned into ℓ subsets called the latest (or L or row) precedence partitions (L_1, L_2, ..., L_ℓ) such that

$$\bigcup_{i=1}^{\ell} L_i = T$$

and $L_i \cap L_j = \varnothing$ when $i \neq j$. L_i represents the subset of tasks that must be executed at least by the end of level or job step i to complete the job in the minimum number of levels ℓ for which all the tasks could be completed.

Given a process with n tasks (T_1, T_2, ..., T_n) whose relationships (dependencies) are indicated by a Single-Exit Connected (SEC) graph with node 1 dummy, we have

Lemma 1: The number of partitions (levels, job steps) in the E-type and L-type partitions are the same for any specific task graph.

Lemma 2: Let $E = (E_1, E_2, ..., E_k)$ and $L = (L_1, L_2, ..., L_k)$ be the E and L partitions of a task graph. Then $E_i \cap L_i \neq \varnothing$ for all $1 \leq i \leq \ell$. In particular, in any SEC graph $E_1 = L_1 = \{1\}$.

Lemma 3: $E_p \cap L_q = \varnothing$ for all $p < q$.

Theorem 1: Given the dependency graph of a task set T, the number of processors needed to compute the job in the least possible time l is bounded below by

$$\max_{\forall j} \left\{ |L_j \cap E_j| \right\}$$

and above by

$$\min\left\{ \max_{\forall j} \left\{ |L_j| \right\}, \max_{\forall j} \left\{ |E_j| \right\} \right\}.$$

Definition: $\lceil y$ is the smallest integer such that $\lceil y \geq y$. $1 \leq x \leq \ell$ is an integer.

Lemma 4: Let k be the minimum number of processors required to process a given task graph G in ℓ+c units of time where c is a nonnegative integer.

Let

$$\max_{\forall x \text{int}; 1 \leq x \leq \ell} \left\{ \frac{1}{x + c} \sum_{j=1}^{j=x} |L_j| \right\} = m.$$

Let

$$M = \lceil m.$$

Then $k \geq M$.

Corollary 1: Given k processors, let t be the minimum time required to process a given task graph G.

Let

$$\max_{\forall x} \left\{ -x + \frac{1}{k} \sum_{j=1}^{j=x} |L_j| \right\} = q$$

Let

$$LBT = \ell + \lceil q.$$

Then $t \geq LBT$. So LBT is a lower bound on time required to process G with k processors.

Corollary 2: Let

$$LBP = \max \left[\max_{\forall j} \left\{ |L_j \cap E_j| \right\}, \max_{\forall x} \left\{ \frac{1}{x} \sum_{j=1}^{j=x} L_j \right\} \right],$$

$$UBP = \min \left[\left(\max_{\forall j} \left\{ |L_j| \right\} \right), \left(\max_{\forall j} \left\{ |E_j| \right\} \right) \right].$$

Then LBP and UBP are lower and upper bounds, respectively, on the minimum number of processors required to process a task graph G with one exit vertex in the smallest possible time (i.e., in ℓ time units).

Algorithms:

There are three algorithms associated with this method. Algorithm A is to determine the minimum number of processors to process a graph in the smallest possible time. Algorithm B is to determine the minimum time required to process a task graph, given k processors. Algorithm C is to determine if a task graph G can be processed in the minimum possible time with k processors. Before describing these algorithms, we have to define some terms.

Definition: P(i) is defined as the set of predecessor tasks of task i and written as $P(i)=\{j \mid j < i\}$. S(i) is defined as the set of successor tasks of task i and written as $S(i)=\{j \mid i < j\}$.

Definition: Let t(i) be the processing time required by task i. A set of tasks $i_1, \ldots, i_k$ are said to be *equivalent* if $P(i_1)=P(i_2)= \ldots =P(i_k)$, $S(i_1)=S(i_2)= \ldots =S(i_k)$ and $t(i_1)=t(i_2)= \ldots =t(i_k)$. A task i *dominates* task j if and only if $P(i) \subseteq P(j)$, $S(i) \supseteq S(j)$, and $t(i)=t(j)$.

Theorem 2: If task i dominates task j, then there exists an optimal solution in which task i is started before or at the same time as task j.

A state in the dynamic program is described by two sets (J,P) where $J = \{j_1, j_2, \ldots, j_q\}$; $j_v \in \{1, 2, \ldots, n\}$ for all v; $P = \{p_1(r_1), p_2(r_2), \ldots, p_u(r_u)\}$; and $P_v \in \{1, 2, \ldots, n\}$ for all v, where n

represents the number of tasks in task graph. The r_v are integers where $0 < r_v < t_{pv}$. The item $P_v(r_v)$ means task P_v requires r_v additional units of processing time to be completely processed at the current time, v=1,2,...,u. State (J,P) at the i-th level represents a set of tasks that are completely or partially processed at the end of the i-th unit of time. The elements of J and P are all vertices of the task graph. A state (J,P) is said to be *infeasible* if there exists tasks a and b such that: 1) $a<b$; 2) $a \notin J$; and 3) $b \in (J \cup P)$.

The remaining task graph R(J,P) of a state (J,p) is the subgraph of the task graph obtained by deleting all vertices in $(J \cup P)$. A state (J,P) is said to be *terminal* if the optimal schedule for processing the remaining task graph R(J,P) has been determined. A nonterminal state (J,P) is said to be *extended* if the successor states of state (J,P) have been constructed. All tasks require equal processing times.

A. Algorithm A

// This algorithm determines the minimum number of processors needed to process a graph in the smallest possible time. //

Initialization: Determine LBP and UBP from the formulas described above;

i-th Step: i = 1, 2, 3, ...

If LBP = UBP, stop;

// the minimum number of processors required is LBP //

If LBP < UBP, use Algorithm C to determine if it is possible to process G in ℓ time units with LBP processors.

If it is possible, stop.

// the minimum number of processors required is LBP //

Else put LBP = LBP + 1; go to the (i+1)th step.

// not possible //

B. Algorithm B

// This algorithm determines the minimum time required to process a graph, given k processors. //

Initialization: Construct one state at the first level.

// this is the state {1} consisting of just task 1. //

Process i-th level: i = 2, 3, ...

1) **IF** all states at the (i-1)th level have been terminated, go to step 2.

ELSE go to step 3.

2) Pick the terminal state with the minimum cost; stop.

// the schedule corresponding to the minimum cost terminal state is the optimal schedule //

3) **IF** all the states in the (i-1)th level have been either terminated or extended, process the (i+1)th level.

ELSE go to step 4.

4) Pick any state S at the (i-1)th level that has not been terminated or extended.

Determine the remaining task graph R(S) for S.

IF R(S) is a tree, go to step 5.

ELSE go to step 6.

5) Determine T(S).

// the minimum time required to process R(S) using Hu's algorithm //

Terminate S with a cost of T(S) + i - 1. Go to step 1.

6) Extend S to the i-th level. Let D(S) = $\{j_1, ..., j_q\}$ be the candidates for processing at the next unit of time.

Construct a successor state to S for each distinct subset $\{i_1, i_2, ..., i_k\}$ of D(S).

There are, therefore, (qk) such successor states.

// we define (qk) = 1 if q < k //

Eliminate all dominated states from further consideration.

Go to step 1.

C. Algorithm C

// This algorithm determines if a task graph G can be processed in the minimum possible time with k processors. //

Initialization: Construct a state {1} at the first level.

// This state consists of just task 1. //

Process i-th level: i = 2, 3, 4, ...

1) **IF** all the states at the (i-1)th level are marked "limit exceeded", stop.

// It is not possible to process the task graph in ℓ units of time. //

2) **IF** all the states at the (i-1)th level are either extended or marked "limit exceeded", process the (i+1)th level.

ELSE go to step 3.

3) Pick any state S at the (i-1)th level that has not been extended or marked "limit exceeded".

Use Corollary 1 to determine LBT(S),

// a lower bound on the time required to completely process the remaining task graph R(S) //

IF LBT(S) > ℓ - (i-1), mark state S with "limit exceeded". Go to step 1.

IF LBT(S) <= ℓ(i-1), go to step 4.

4) **IF** R(S) is a tree in which all tasks have equal time, go to step 5.

ELSE go to step 6.

5) Determine t(s),

// the minimum time required to process R(S) //

IF R(S) is a tree with all tasks of equal processing time, it can be shown that t(s) = LBT(S). Hence t(s) <= ℓ - (i-1); stop.

// there does exist a schedule for processing the task graph G in ℓ units of time //

6) Extend S to the i-th level. Let D(S) = {j_1, ..., j_q} be the candidates for processing at the next unit of time.

Construct a successor state to S for each distinct subset {i_1, i_2, ..., i_k} of D(S).

Eliminate all dominated and infeasible states from further consideration. Go to step 1.

Validation:

The efficiency of the algorithm clearly depends on the "tightness" of the bounds LBP and UBP [Ref. 37: P. 140]. [Ref. 37] indicates that these bounds are quite tight; however, [Ref. 10] gives lower and upper bounds for the minimum number of processors and to a lower bound for the minimum time, which are sharper than these values, and should give a more efficient version of this algorithm.

III. DESIGN OF OPTIMAL STATIC SCHEDULING ALGORITHMS

We introduce three algorithms in this Chapter. In Section A, we design an algorithm called CP/MASPF (Critical Path/Most Accumulated Successive Paths First). There are many tasks in the task graph to be scheduled. There are precedence constraints on the tasks; however, the timing constraints are not considered. In such a case, the purpose of scheduling is to find an algorithm to execute all the tasks in the graph as soon as possible.

In the rest of this Chapter, starting from Section B, we consider scheduling the tasks in PSDL. Tasks in PSDL have timing constraints associated with them, such as earliest starting times and deadlines. In this case, the purpose of scheduling is not to finish the task graph as soon as possible but rather to finish them with the least tardiness.

Scheduling PSDL tasks in multiprocessor systems can be described as the following steps:

1. Construct a graph of constraints, and
2. Find a possible schedule from the graph.

In Section B, we describe the PSDL operators' characteristics. In Section C, we describe a way to construct the constraints graph of tasks. In Section D, we discuss all the timing properties. Any sequence which is feasible must not violate any timing constraint. We provide two heuristics to solve PSDL scheduling on multiprocessors in Section E. In Section F, we provide two techniques to search for the optimal solution.

A. THE CRITICAL PATH/MOST ACCUMULATED SUCCESSIVE PATHS FIRST ALGORITHM (CP/MASPF)

1. Preliminary Problem Description

We consider a scheduling problem which can be formulated as follows. There are n tasks with known times to perform each task and with technological ordering restrictions among the tasks. Each processor can execute any task; however, each task can be executed on only one processor at a time. We assume that the task is non-preemptive. There are two questions to answer.

1. Assuming that all tasks must be completed by time t, find a schedule that requires the minimum number of processors. It is assumed that all the processors are of equal ability and each of them can do any of the n tasks.

2. If m processors are available, arrange a schedule that completes all tasks at the earliest possible time.

The critical path method is the most efficient technique to solve such problems. This algorithm is described in Chapter two. This method was first proposed by Hu [Ref. 18]; however, this was a simplified version in which all tasks require equal time. The tasks do not necessarily have unit processing time in [Ref. 37]. They solve the problems by using upper and lower bounds on both processors and total execution time.

Hu assumes that the choice is arbitrary when the priority is equal among those tasks. Since the priority order cannot be determined uniquely when there exists a plurality of tasks having

one and the same level, there still exist some open problems. In [Ref. 21], this approach is modified so that the task having the largest number of immediately successive tasks is assigned the highest priority. This method is called the CP/MISF (critical path/most immediate successors first).

In our algorithm, we will assume that tasks can have different processing time; in such a case, the processing time of a task is the sum of unit processing times. If task T_i requires t_i units of processing time ($t_i>1$), we replace task T_i by t_i tasks which require one unit of processing time.

We introduce a heuristic method called Critical Path/Most Accumulated Successive Paths First (CP/MASPF). This method is similar to Critical Path method; however, it exploits the priorities of tasks which have the same level. The CP/MISF method also introduces such priorities, but focuses on the immediate successors and then loses some power.

In Hu's algorithm, the choice is arbitrary if two or more tasks have the same priority. The CP/MISF improves Hu's algorithm by solving the competition of these tasks . The CP/MASPF method also solve this problem but using different approach.

2. Description of CP/MASPF Algorithm

We label each node in the task graph with a pair of integers. The first element of the pair represents the level of the node in the task graph. The second element of the pair represents the number of paths accumulated after such node.

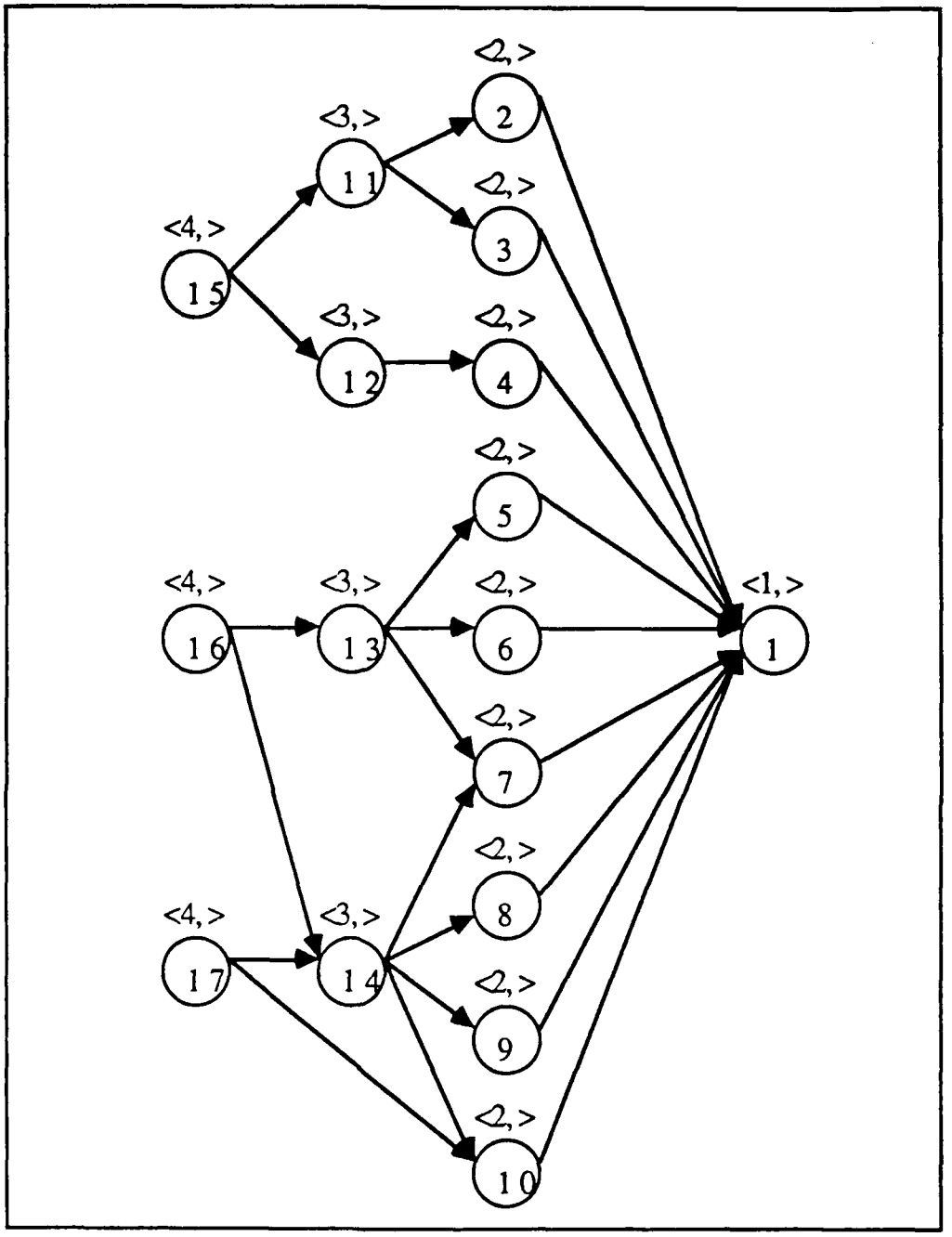


Figure 16 Task Labelling of the Graph as the First Element

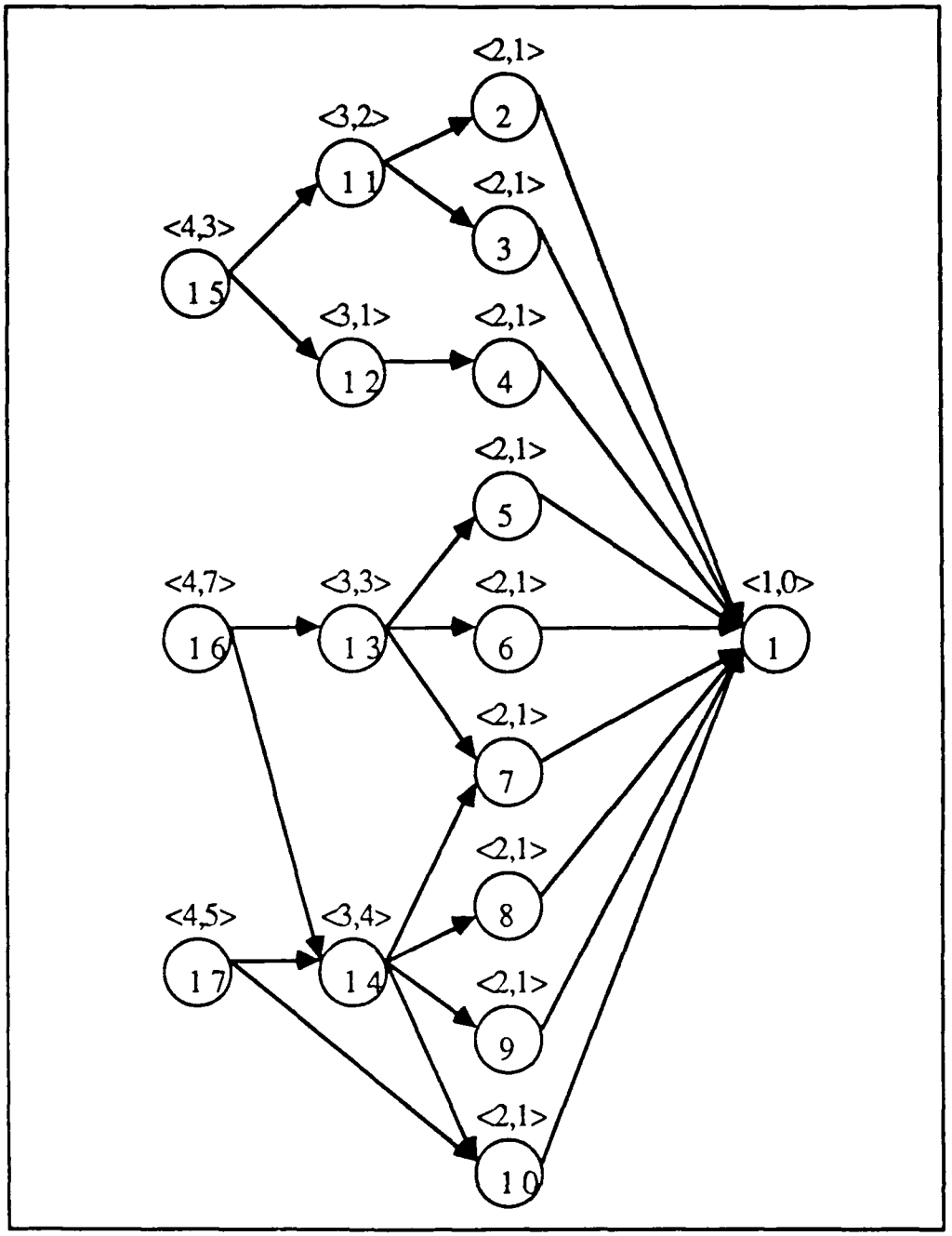


Figure 17 Task Labelling and Path Calculation as the Two Elements of Nodes of the Graph

In the first step of task labelling, we assign a level number to each node which represents a task in task graph G. This graph contains a dummy final node which represents the end of the schedule. A node Ni is labelled with $\alpha_i = x_i + 1$ if x_i is the length of the longest path from Ni to the final node in G. The final node is labelled 1. Figure 16 on page 58 illustrates the task labelling of the task graph. The number inside the circle represents the task number. In Figure 16, the path length from node 17 to node 1 is 3; so the level of node 17 is then $\alpha_{17} = 3+1 = 4$. Nodes 11, 12, 13, and 14 have the same level which is three.

Figure 17 on page 59 illustrates the second step of the method, which is the calculation of accumulated path number between the current node and the final node. The number of accumulated paths successive to node i is denoted as P_i. In Figure 17, node 2 has only one path to the final node, so the accumulated path number is 1. Node 16 has two paths to the successive nodes which are node 13 and 14. These nodes have three and four directly successive nodes respectively. The accumulated path number of node 16 is then $P_{16} = 3+4 = 7$.

It is very easy to calculate the accumulated successive path number of each node. We start with the final node and trace backwards. All the nodes labelled with <2, > have only one path to the final node. In such a case, $P_i = 1$ for all nodes i with $\alpha_i = 2$. It is important to note that we assume only one final node exists. If node i has n immediate successors, the accumulated successive path

number is the sum of the accumulated path numbers of the direct successor nodes.

We can calculate the number of the accumulated paths successive to each node backwards as well as the level number for each node. This means we can do both these two things with just one pass through the task graph.

The scheduling algorithm uses the labelled task graph as follows. If the number of available processors is greater than or equal to the number of current starting nodes, assign all of these tasks to available processors.

If the total number of current starting nodes is greater than the number m of available processors, choose m current starting nodes with the highest priorities, where priorities are determined as follows. If two tasks have different level numbers, the one with a larger level number has a higher priority. If the tasks have the same level values, those tasks which have more successive paths (P_i) have higher priorities than those with fewer successive paths.

The CP/MASPF method consists of the following steps.

Step 1: Determine the level α_i as well as the accumulated successive paths P_i for each task.

Step 2: Construct the priority list in the descending order of α_i and p_i for nodes without predecessors. If tasks have different level values, then tasks with higher values have higher priorities. If tasks have same values, then tasks with higher P_i values have higher priorities.

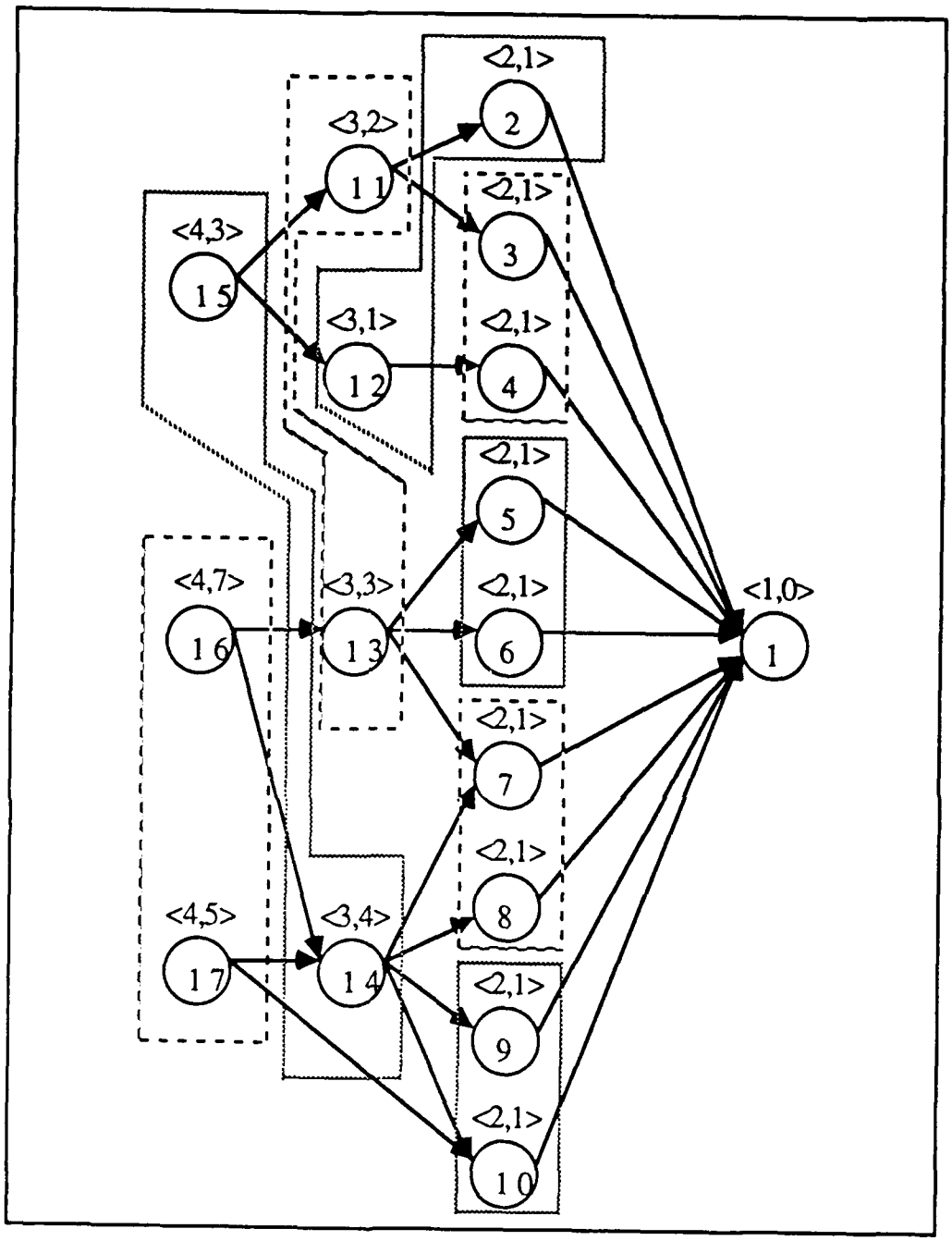


Figure 18 Task Assigning for Two Processors

Step 3: Execute list scheduling on the basis of the priority list. When a task completes, its node is removed from the graph. Any

nodes whose only predecessor is removed from the graph are added to the priority list in the proper order. Removing a node without any predecessors does not change the level number or successive path number of any other node in the graph.

Figure 18 on page 62 gives an example of this algorithm. Given Figure 17 and there are only two processors available, Figure 18 shows how to choose the tasks. Since we have only two processors, we should choose two nodes among nodes 15, 16, and 17. These three nodes have the same values as the first elements in their integer pairs.

According to the algorithm, we should compare the second atoms of the three integer pairs. Node 15, 16, and 17 have the values 3, 7, and 5 respectively. We then choose nodes 16 and 17 first. This rule is repeated for the remaining graph until all the nodes are chosen.

It is important that task labelling and accumulated path number calculating can be done in parallel. Thus the calculation of the number of accumulated paths does not influence the computational complexity. Hu had shown that CP method completes all jobs at the earliest time [Ref. 18: PP. 846-848]. The CP/MASPF method is based on CP method and solves the problem of same priority tasks heuristically, which is not discussed in [Ref. 18]. As a result, the CP/MASPF method is also optimal.

In order to compare the deviation, consider Figure 17 as an example. Suppose we have a task graph illustrated in Figure 17 and

two processors. By using Hu's algorithm, nodes 15, 16, and 17 have the same priority which is 4. Therefore, two of them are chosen randomly. By applying CP/MISF method, nodes 15 and 16 have two immediate successors which have the second high priority. Although node 17 has two successors, only one of them has second high priority. The choice is then nodes 15 and 16 first. If we apply CP/MASPF method, nodes 16 and 17 are chosen first. It is reasonable to choose nodes 16 and 17 first because there are more nodes which cannot be processed without finishing these two nodes.

B. OPERATORS AND TASKS

The PSDL language is based on a computation model which treats software systems as networks of operators communicating via data streams. As we mention in Chapter one, this model is an augmented directed graph G = (V, E, TC(v), C(v)), where V is the set of vertices, E is the set of edges, TC(v) is the set of timing constraints for each vertex v, and C(v) is the set of control constraints for each vertex v.

The definition of PSDL operators is in Chapter one. We consider only periodic PSDL operators in our scheduling algorithms. Each periodic operator appears more than once during the execution of the problem. We define a task as each instance of any operator.

Each periodic operator in PSDL is time-critical; namely, an operator with at least on timing constraint associated with it. Some of the timing constraints are defined in Chapter two. In PSDL, T(v) is the set of timing constraints of each operator. The timing constraints can be divided into two subsets: the original constraints and the

derived constraints. The original PSDL timing constraints contain the following attributes:

1. MET(k) : Operator k requires at most MET(k) time units of processing.

2. PERIOD(k) : Period of the operator k.

3. FINISH_WITHIN(k) : Maximum time allowed to finish operator k after the earliest start.

The derived PSDL timing constraints contain the following attributes:

1. PHASE(i) : Phase of the base operator of task i.

2. INSTANCE(i) : The rank of task i in the sequence of instances of the base operator. The sequence is ordered by starting time, and the first instance has instance number zero.

3. EARLIEST_START(i) : Earliest possible starting time for task i.

4. DEADLINE(i) : The latest time by which task i must be completed.

5. COMPLETION(i) : Time when the task i is actually completed.

6. TARDINESS(i) : The amount of time by which i missed its deadline, negative if task i completed before its deadline.

7. NUMBER_OF_INSTANCES(k) : The number of instances of operator k in the harmonic block.

In order to compare MET, FINISH_WITHIN, and deadline of operator, Figure 19 on page 66 shows a modification of Figure 8. The derived timing constraints are not defined by the user, they are

determined by the scheduling algorithm during the analysis of the problem.

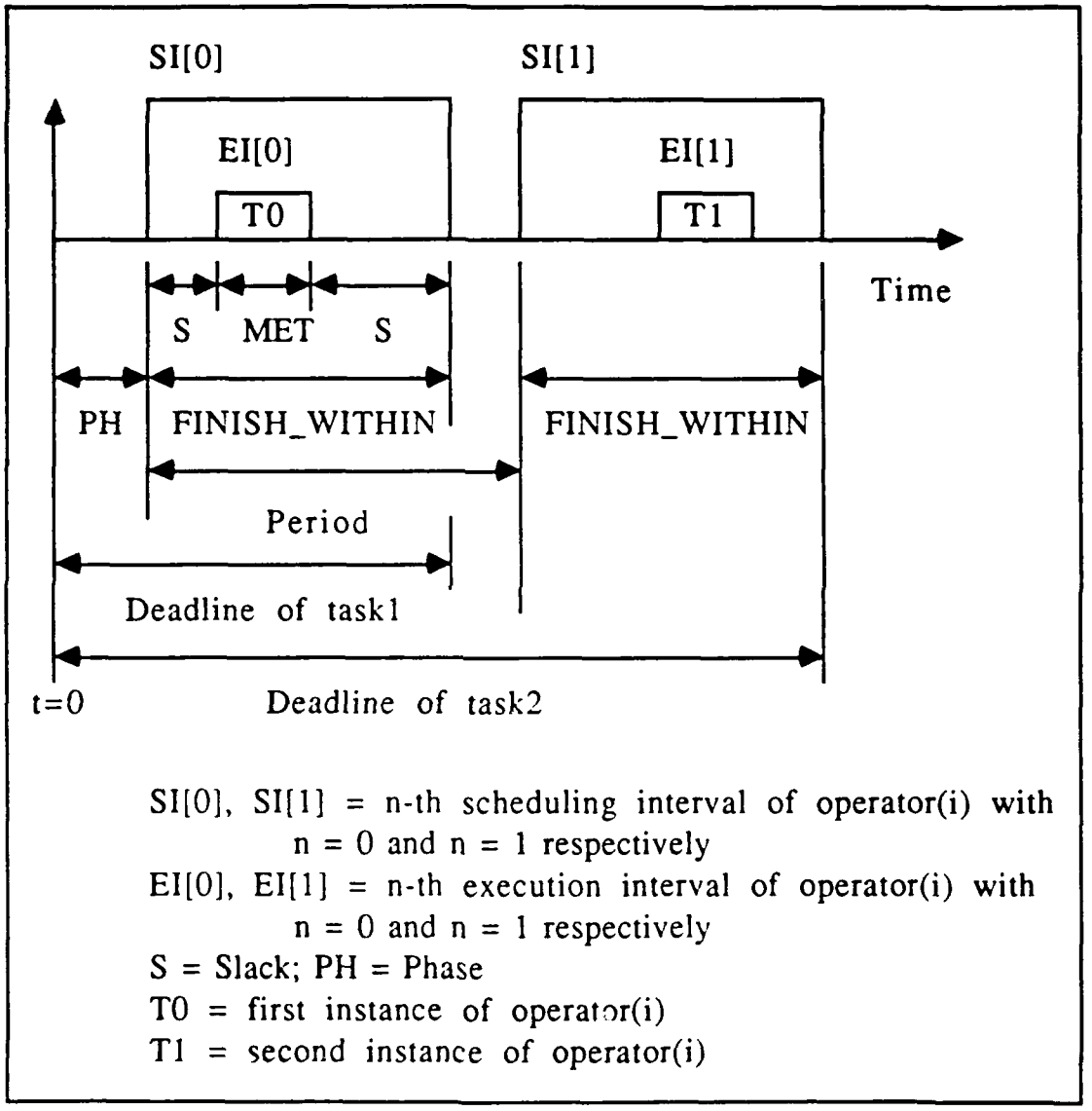


Figure 19 Midified Timing Constraints Diagram for a Periodic Task

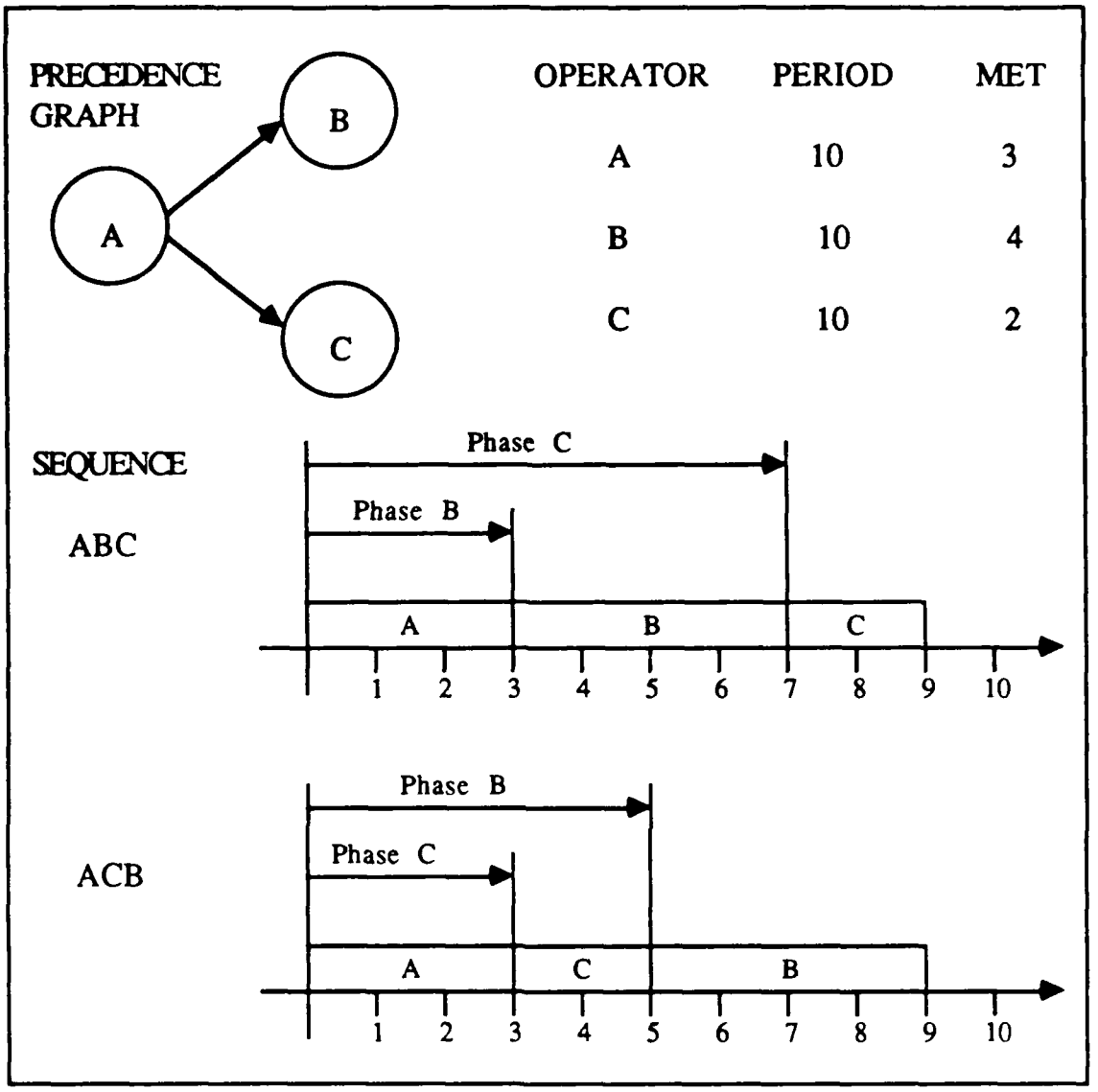


Figure 20 Possible Phases of an Operator

Each operator in PSDL has a phase. The phase of the task is defined as the delay between the reference time zero and the starting time of the first scheduling interval for this operator, $0 \leq$ phase $\leq$ period. The phase is a function of the operator and the

permutation of operators chosen. Figure 20 on page 67 illustrates the relationship [Ref. 5].

To simplify further manipulations on the data, two dummy operators are included in the set of vertices V: the dummy operator V(0) and the dummy operator V(n+1), where n is the number of operators in the original set V furnished by the user. Since V(0) and V(n+1) are dummy operators, the original set of operators is counted from V(1) to V(n).

Each atomic operator i in the original set of operators should meet the following timing constraint:

TC(i).MET ≤ TC(i).FINISH_WITHIN

This constraint does not apply to the two dummy operators.

C. CONSTRAINT GRAPH OF PSDL OPERATORS

1. Description of the Steps to Obtain the Graph of Constraints

The graph of constraints shows all of the tasks in a repeating harmonic block, which will be the basis for the static schedule. Each operator corresponds to one or more tasks in the graph of constraints. Each task is an instance of the associated operator. The number of tasks depends on the ratio of the period of the harmonic block to the period of the operator.

The graph of constraints is completely defined and evaluated using the algorithms described in the following subsections. There are six steps to generate a DFD of the global algorithm to generate the graph of constraints [Ref. 5].

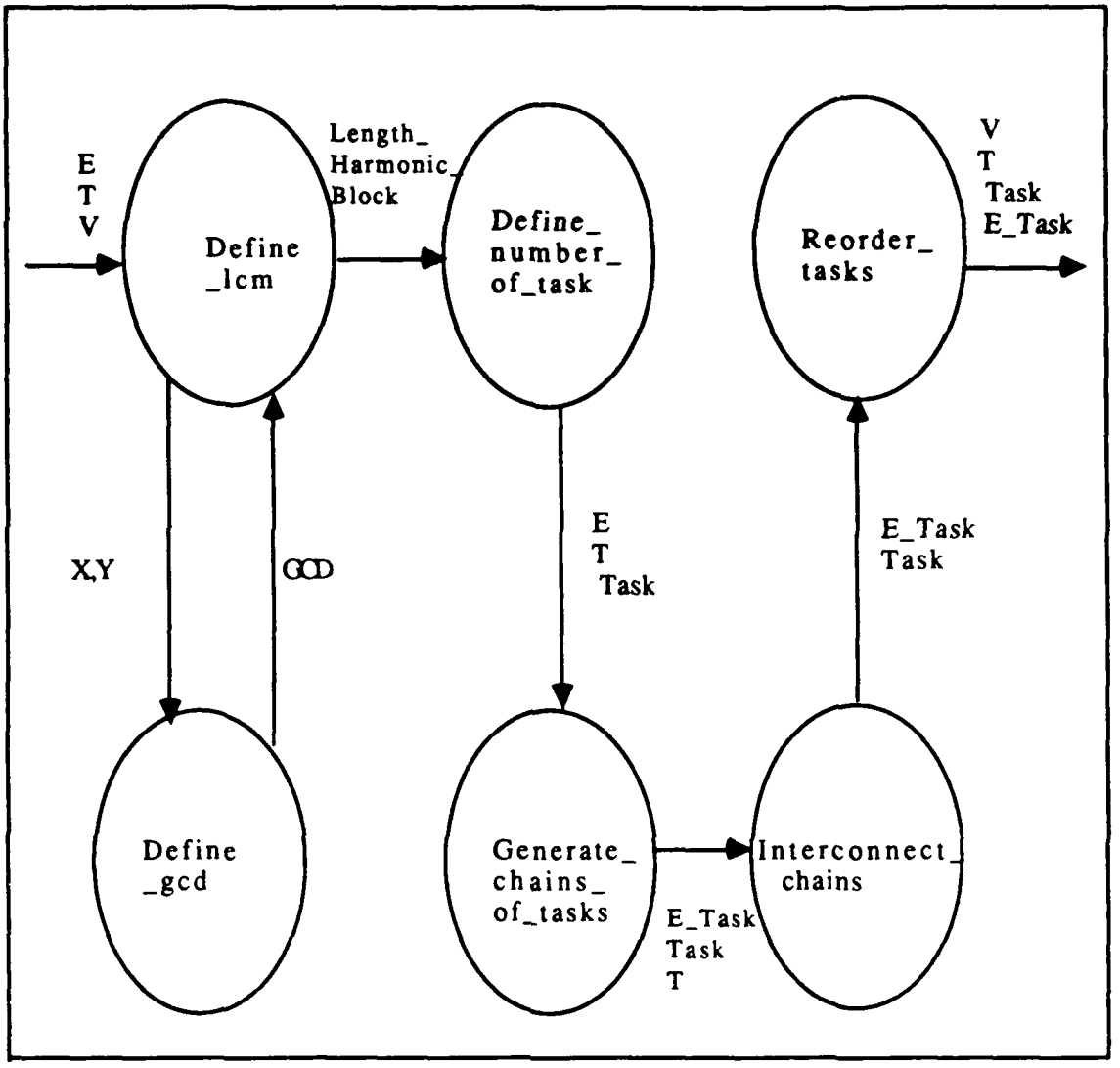


Figure 21 The First Level DFD Graph of Constraints

1. Evaluation of the GCD of the operators,
2. Evaluation of the LCM of the operators,
3. Evaluation of the number of tasks in the graph of constraints,
4. Generation of chains of tasks,
5. Interconnection of the chains, and
6. Reordering the graph of constraints.

Figure 21 on page 69 shows the first level of DFD.

Figure 21 on page 69 shows the first level of DFD.

2. Length of the Harmonic Block

In order to obtain the graph of constraints, we need to define a time frame. The approach that we selected is to define a harmonic block as described in [Ref. 35: pp. 34-41]. This harmonic block, if repeated in time, ensures that all the time-critical operators are performed within their timing constraints. This means we will map the harmonic block to our constraint graph.

The length of the harmonic block is simply the least common multiple (LCM) of all the operators that belong to the set in analysis. The LCM is computed by taking two periods at a time, multiplying them together, and then dividing this result by the greatest common divisor (GCD) of the two periods. This result is then multiplied together with the next period and divided by their GCD until all operators in the set have been processed. The result of this operation on the last pair in the set is the LCM of all operators in the set. The algorithm to find GCD is in [Ref. 35: p. 37]. The algorithm to find LCM is in [Ref. 35: p. 41].

3. Tasks in the Graph of Constraints

The tasks are the instances of each operator that must be executed inside the time frame. The number of tasks for each operator is obtained by dividing the time length of the harmonic block by the period of the corresponding operator. The result of this operation is stored in the derived timing constraint records of TC(v) for each operator.

ALGORITHM FOR NUMBER OF TASKS

N_MAX := number of operators in the set V, LCM as defined before;

For N in 1 .. (N_MAX) loop

// need not consider dummy operators //

TC(N).NUMBER_OF_TASKS := LCM / TC(N).PERIOD;

end loop;

// End of algorithm. //

Example 1: Suppose we have the following data:

Operator	period	
0	-	(dummy operator)
1	10	
2	4	
3	6	
4	-	(dummy operator)

then we have GCD = 2 and LCM = 2*5*2*3 = 60.

Step 1: N_MAX = 3, LCM = 60.

Step 2: N = 1, T(1).NUMBER_OF_TASKS = 60 / 10 = 6.

Step 2: N = 2, T(2).NUMBER_OF_TASKS = 60 / 4 = 15.

Step 2: N = 3, T(3).NUMBER_OF_TASKS = 60 / 6 = 10.

4. Precedence Constraints of the Tasks

The generation of the graph of constraints of the tasks is done in two steps. During the first step we produce a partially ordered set of tasks for each operator, and in the second step we use

the precedence constraints among the operators to generate the precedence constraints among the tasks.

The tasks for each operator are ordered according to the principles explained next. We can describe precedence relationships using a mathematical formula as well as an abstract data type. Let i and j be two different tasks. The precedence relationships among the set of tasks for a single operator are defined by the following.

$i \prec j$ iff phase(k) + period * i + finish_within

$\leq$ phase(j) + period * j;

where i and j are instance numbers.

The precedence relation for the instances of an operator is a single chain if finish_within $\leq$ period, and has multiple chains otherwise.

A partially ordered set (poset) is called a chain if exactly one permutation is feasible (is consistent with the partial ordering) [Ref. 5].

We can also define the precedence relationship as a boolean operation on the precedence-relation data type with the following interface:

Function precedes (i, j: node; R: precedence_relation)

return boolean;

-- True if $i \prec j$.

The interconnection of chains of tasks is based on the following relation. Suppose O_1, O_2 are operators and T_1, T_2 are

corresponding tasks with instance numbers I_1, I_2 and with periods P_1, P_2. We have

TASK(T_1).OPERATOR_NUMBER = O_1,

TASK(T_2).OPERATOR_NUMBER = O_2,

TASK(T_1).INSTANCE_NUMBER = I_1,

TASK(T_2).INSTANCE_NUMBER = I_2,

TC(O_1).PERIOD = P_1, and

TC(O_2).PERIOD = P_2.

We generate the precedence relations (graph of constraints) via the following operation:

```
Function task_edge(O1, O2: operator; T1, T2: task)
        return boolean
        -- True if there is an edge from O1 to O2 in the PSDL
        -- graph G and P1 * I1 = P2 * I2.
```

The first instance of each operator i is preceded by the dummy operator 0 if and only if there is no other task that precedes i, and the last instance of each operator i precedes the dummy operator N+1 if and only if it does not precede another task.

The elements TASK(0) and TASK(TASK_LENGTH+1) (the only instances of the dummy operators V(0) and V(N+1), respectively) are dummy tasks used in the construction of the graph of constraints.

Algorithms for generating chains of tasks and for interconnecting chains of tasks are available in [Ref. 5: PP. 83-87]. We only introduce an example used in [Ref. 5].

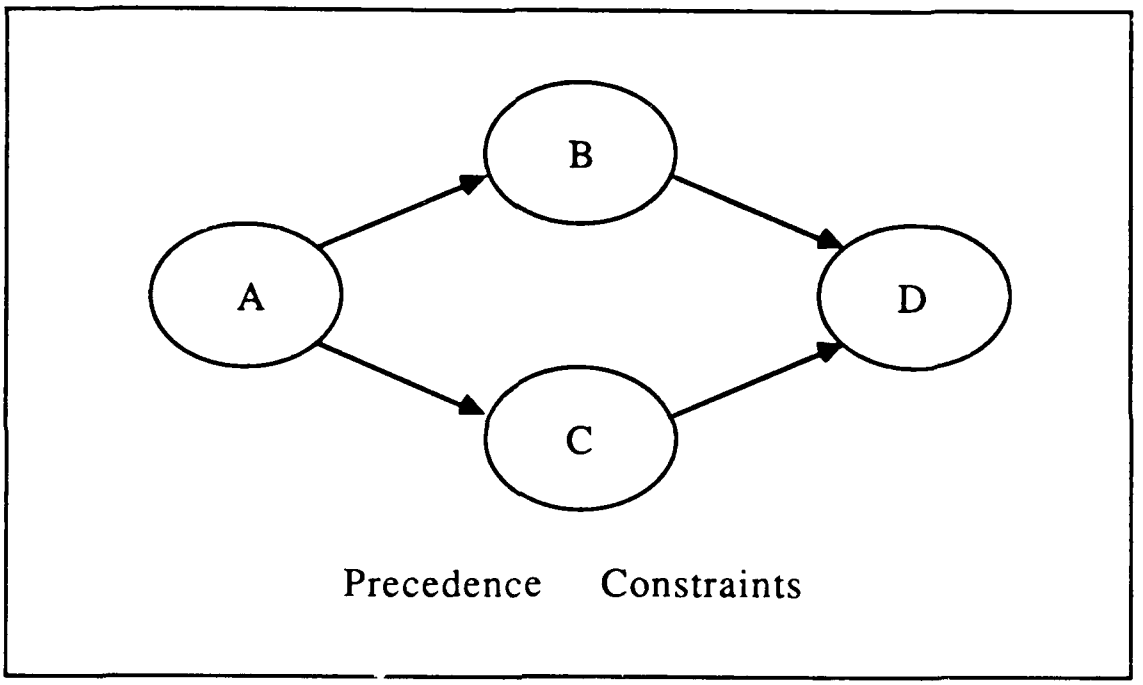
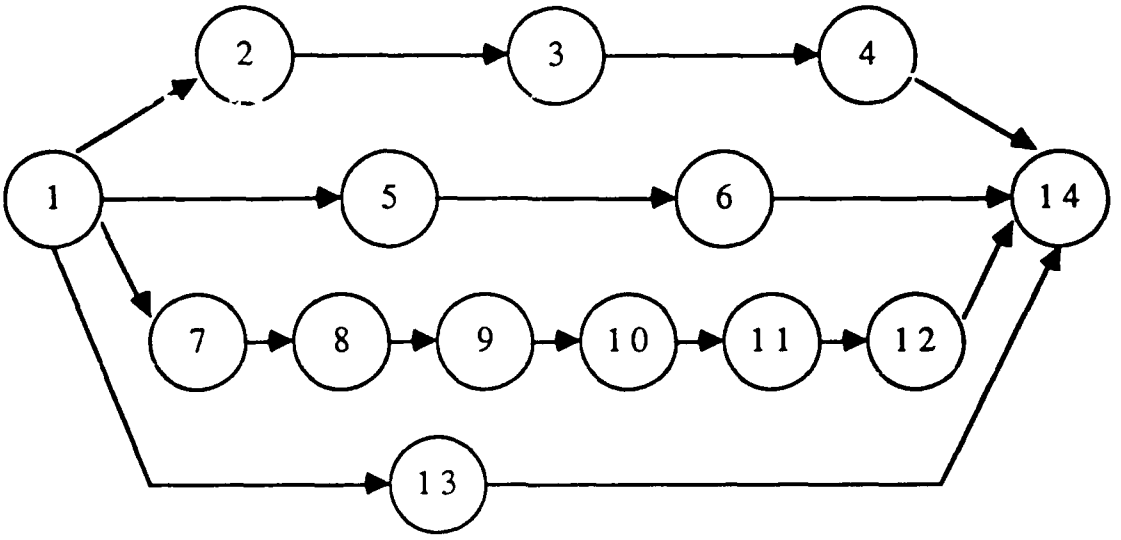


Figure 22 Precedence Constraints

Suppose we have an operator graph with precedence constraints as illustrated in Figure 22 on page 74. The operators A, B, C, D have periods of 10, 15, 5, 30 time units respectively and finish_within = period for all the operators. After applying the algorithms of LCM and number of tasks, the data available for the set V and T is the following:

i	V(i)	T(i).PERIOD	LCM	T(i).NUMBER_OF_TASKS
0,5	dummy	-	-	-
1	A	10	30	3
2	B	15	30	2
3	C	5	30	6
4	D	30	30	1



Graph of Constraints

Task	OP	Instance
1	-	-
2	1	0
3	1	1
4	1	2
5	2	0
6	2	1
7	3	0
8	3	1
9	3	2
10	3	3
11	3	4
12	3	5
13	4	0
14	-	-

Task Description

Figure 23 Chains of Tasks

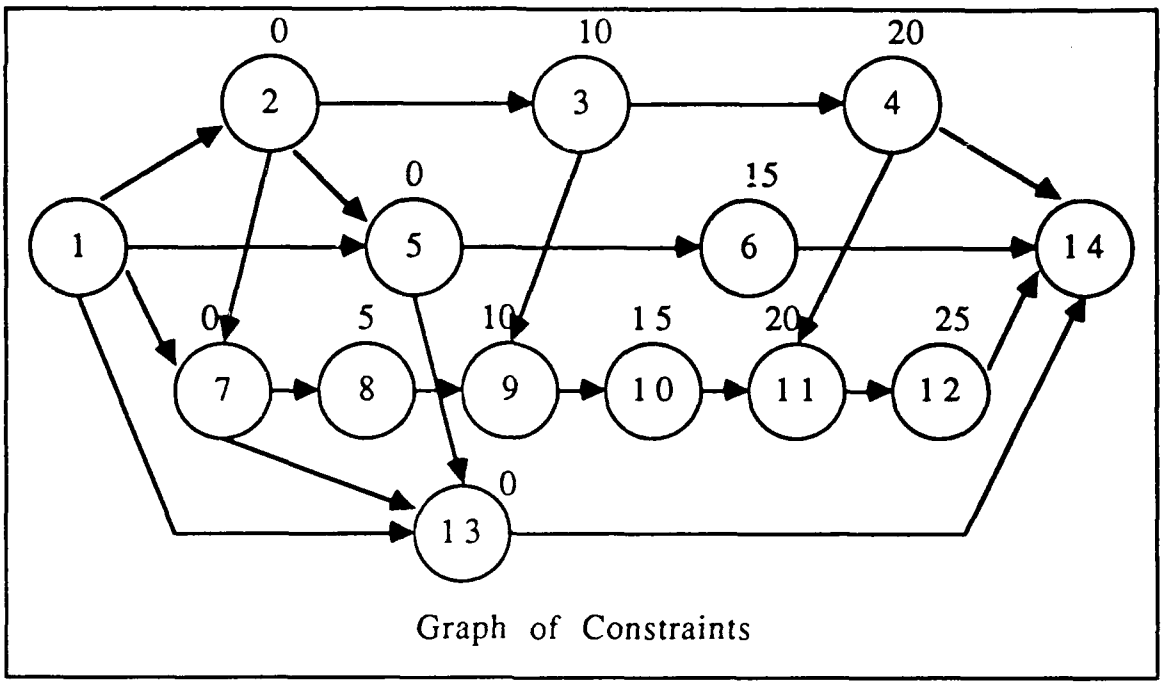


Figure 24 Graph of Constraints

In this case we have a single chain for each operator because finish_within ≤ period. Using the algorithm to generate chains of tasks, we can get Figure 23 on page 75, and an application of the algorithm to interconnect chains of tasks is illustrated in Figure 24 on page 76.

Suppose there are m processors. After we create the graph with interconnecting chains, we can schedule all the tasks in graph to these m processors. Heuristics for doing this are presented in Section E.

D. COST FUNCTION

The performance objective of meeting task deadlines is the scheduling criterion applicable to PSDL. All periodic PSDL operators

have earliest times as well as deadlines. The most important goal in scheduling the tasks is that the resulting schedule should meet these timing constraints.

We need to introduce some concepts for the analysis and evaluation of cost functions on a set of sequences S = {s(1), ..., s(m)} which contain tasks. Each sequence s(m) = [s(m,1), ..., s(m,n)] represents the sequence of tasks performed by the m-th processor, in the order in which they are performed.

The cost function assigns an integer value (or cost) to each set of sequences. There are many ways for us to assign n tasks to m processors. Each processor has one and only one sequence of tasks. This sequence is a subset of n tasks. The input to the cost function is a set of m sequences. The cost function calculates the maximum tardiness of the best schedule with the given processor assignments and task sequences.

For each sequence assigned to a particular processor, we define this sequence as legal if it satisfies the precedence constraints represented by the graph of constraints, and as feasible if and only if it satisfies simultaneously the precedence constraints represented by the graph of constraints and the timing constraints.

The tasks in each feasible sequence being evaluated obey the following equations:

(1). Tardiness(i) = Completion(i) - Deadline(i)

(2). Completion(i) = Start(i) + MET(OP(i))

(3). Start(i) = max { Completion(s[p(i), q(i)-1]), Earliest_Start(i) },

where p(i) is the processor identifier of task i and q(i) is the position in s of task i.

(4). s[m,0] = 0, Completion(0) = 0

(5). Earliest_Start(i) = Phase (OP(i)) + Instance(i) * Period(OP(i))

(6). Phase(OP(i)) = Earliest_Start(j),

where OP(j) = OP(i) and Instance(j) = 0

(7). Phase (OP(i)) ≤ Start(j) - [Instance(j) * Period(OP(j))]

for all j such that OP(j) = OP(i)

(8). Deadline(i) = min { Earliest_Start(i) + Finish_within(OP(i)),

Length_of_harmonic_block + earliest_start(p) }

if task i is scheduled on processor p.

To check if a legal sequence is feasible is to check if all the tasks in the sequence meet the tardiness constraints. To check if the multischedule for multiprocessor is feasible is to check if all the sequences corresponding to these processors are feasible. If at least one task in a sequence cannot meet the constraint then we say that this schedule is not feasible. If at least one sequence in a multischedule is not feasible, we say this multischedule is not feasible. If there is no feasible multischedule, we have to find an optimal one. Section F will offer some methods to solve this case.

The algorithm for evaluating the cost is illustrated as follows:

```
Evaluate_sequences(number_of_processors: in integer;
                   multischedule: in set{sequence{task}};
                   cost: out integer; feasible: out boolean) is
begin
```

```
    cost := infinity;
    for each sequence s in multischedule loop
            cost := max(cost, tardiness(t));
           if tardiness(t(j)) > 0 then
             feasible := false;
           end if;
    end loop;
end evaluate_sequences;
```

E. HEURISTICS FOR ASSIGNING TASKS TO PROCESSORS

As we mentioned in Section D, there are many ways for us to assign n tasks to m processors. There are two questions derived from this point: what order should the processor do the task and which processor should do which task. Algorithm A provides a heuristic for solving these two questions. One way to solve the competition between tasks is most-urgent first-serve, which is also called earliest task deadline/processor ending time first and described as follows:

1. Algorithm A: Earliest Task Deadline/Latest Processor Ending Time First

```
create a task graph with interconnecting chains by using the
algorithm described in Section C;
remove the dummy nodes 0 and n+1;
while the task graph is not empty loop
  find the set of nodes S which have no ancestors.
  choose task t from S with the earliest deadline;
  choose processor P with the latest ending time such that
```

```
        ending_time(P) ≤ deadline(t) - met(t);
      schedule task t on processor P;
      remove task t from the task graph;
    end loop;
    evaluate_sequences(number_of_processors, tasks_processed);
      // check the cost function (defined in Section D of this Chapter)
           to see if the sequences are feasible //
end Algorithm A;
```

The reason to assign earliest deadline task with highest priority is to ensure that task can meet its deadline as well as possible. The reason to choose the processor with the latest feasible ending time is to minimize the processor idle time. We provide another heuristic next.

2. Algorithm B:

Earliest Task Start Time/Most Available Processor First

```
multischedule(number_of_processors: in integer; TASK: in {task};
                    multischedule: out {sequence{task}}) is
begin
  while TASK is not empty loop
        t(j) := j | for all x, start(j) <= start(x)
        -- select task which has earliest start time
        remove(t(j), {task});
        if {p(i) | ending time of p(i) <= earliest start t(j)} is not
              empty then
             select p(i) with the latest ending time such that
```

```
                ending time <= earliest start;
                -- otherwise it will have more idle time for
                -- processor
            else
                select p(i) with the earliest ending time;
                -- shortest time for t(j) to wait
            end if;
        end loop;
        schedule task t(j) on processor P(i);
        evaluate_sequences(multischedule,number_of_processors);
    end multischedule;
```

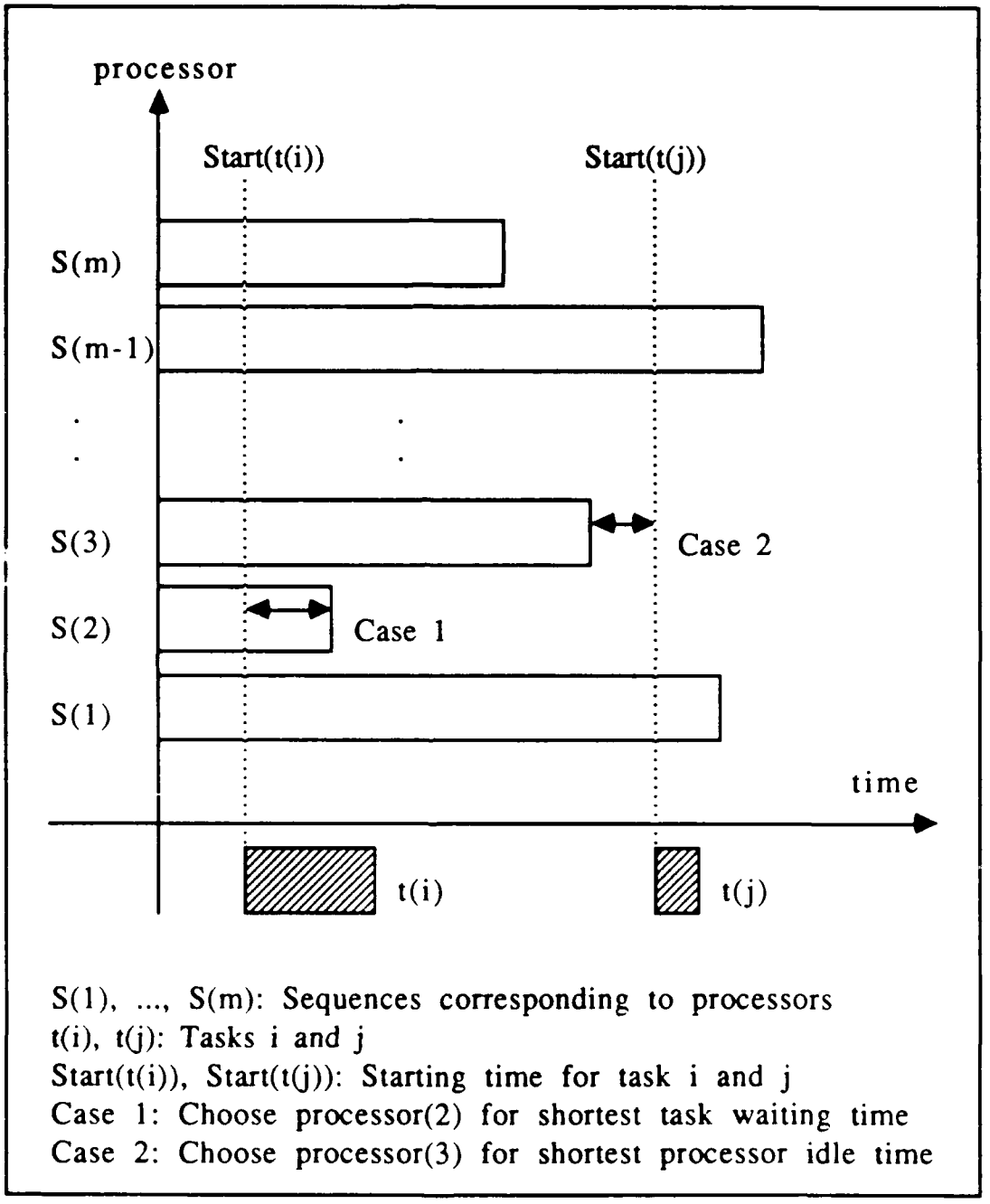
Figure 25 on page 82 illustrates the assignment of tasks using Algorithm B. Figure 25 gives two critical cases:

1. Task comes before any processor is ready, and
2. Task comes after at least one processor which is ready.

By applying our algorithm, case 1 will choose processor(2) and case 2 will choose processor(3).

F. FINDING THE OPTIMAL SOLUTION

One way to find if there exists optimal solution for multiprocessor scheduling is to search all the possible nodes in the multischedule space graph. There are two attributes in the multischedule space graph: nodes and edges. The nodes in the graph represent tasks to be scheduled. The edges in the graph represent how we assign nodes to processors. The first algorithm to search for the optimal solution is described as follows:



S(1), ..., S(m): Sequences corresponding to processors
t(i), t(j): Tasks i and j
Start(t(i)), Start(t(j)): Starting time for task i and j
Case 1: Choose processor(2) for shortest task waiting time
Case 2: Choose processor(3) for shortest processor idle time

Figure 25 Example for Tasks to Choose Processors

```
find_multischedule(number_of_processors: in integer; cost: out
                        integer; best_sequence: out {sequence{task}}) is
cost := infinity;
begin
  for each possible set of sequences ms
       if tardiness(ms) < cost then
        -- ms means multiple sequences
        cost := tardiness(ms);
        best_sequence := ms;
      end if;
  end loop;
end find_multischedule;
```

The tardiness function can be calculated using the evaluate_sequences procedure given in Section C. The algorithm described above is very slow. It supports a way to traverse all possible combinations and then finds a best one. A better method to find an optimal solution is to use the branch_and_bound technique. This method can help us avoid expanding those nodes whose estimate values are worse than current one. The algorithm is illustrated as follows:

```
find_multischedule(g: graph; multischedule: out {sequence{task}};
                        feasible : out boolean;
                        best_schedule: out set{sequence{task}}) is
best_cost : integer := infinity;
begin
```

```
    branch_and_bound(g, multischedule, best_cost, best_schedule);
   feasible := (best_cost <= 0);
end  find_multischedule;

branch_and_bound(g: graph; ms: {sequence{task}};
                        best_cost: in out integer;
                        best_schedule: out {sequence{task}}) is
begin
     if g is empty and cost(ms) < best_cost then
       best_cost := cost(ms);
     end if;
     for each node n in g such that predecessors(n) = { } loop
        case i in [1 .. number_of_processors]
            if max{cost(append(n,ms,i),
                    least_cost(g-n, append(n,ms,i))} <
              -- node n appended to sequence i of ms
            best_cost then
              branch_and_bound(g-n, ms(i) || [n], best_cost);
            end if;
        end case;
        if best_cost <= 0 then return; end if;
     end loop;
end  branch_and_bound;

least_cost(g, ms) =
```

```
    max n: node in g of
            latest_ending_time(ms) - deadline(n) +
           ( sum(MET(m) such that m in ancestors(n) and not m in
             ms(i)) where i in [1 .. number_of_processors] )
           / (number of processors);
end least_cost;
```

IV. CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE WORK

A. SUMMARY

This thesis provides an introduction to real-time systems which is different from traditional computer systems. There are two main software engineering methodologies, the traditional life cycle and rapid prototyping. In particular, the rapid prototyping methodology was discussed as a pioneering approach to the development of software more efficiently and at less cost and risk. Rapid prototyping is supported by a particular language called Prototype System Description Language. The Computer Aided Prototyping System (CAPS) was introduced as a software engineering tool that is currently being designed. This tool will enable software designers to exploit rapid prototyping to its fullest by automating the construction of executable prototypes. The execution support system is the component within the CAPS which makes the PSDL language executable. The major contribution of CAPS to the advancement of software engineering technology lies in the fact that the executable prototypes can be automatically generated by the use of specifications and reusable software components.

We define some scheduling terms and basic concepts needed to schedule tasks in multiprocessor systems. We survey previous research in this field. The harmonic block with precedence

constraints scheduling algorithm is used by CAPS. The idea of harmonic block is adapted in this thesis. The critical path method is a kind of heuristic. Originally it was created for the tasks which have only a precedence relationship. It can also be applied in task systems with timing constraints. The critical path method with upper bound and/or lower bound algorithm is an extension of the critical path method. It uses another approach to show that there exists bounds in the scheduling algorithm. The efficiency of the algorithm depends on the tightness of the bounds. The sharper the bounds are, the more efficient this method is.

There is no polynomial algorithm for optimal multiprocessor scheduling with deadlines and precedence constraints found so far. Even if we prove that these problems have no polynomial time algorithms, we still have to solve them. We can simplify the problems or use heuristic methods to approach practical solutions.

We design three algorithms for different situations. We design an algorithm called Critical Path/Most Accumulated Successive Paths First (CP/MASPF). We suppose that there are no timing constraints in this method. The scheduling goal is to create a schedule as soon as possible, meeting the precedence constraints. The deviation from other research is that it solves the competition of the same priority tasks, using accumulated paths as a measure.

Tasks in PSDL have timing constraints associated with them. The goal of scheduling PSDL tasks is to create a schedule as soon as possible, meeting both the precedence constraints and timing

constraints. We describe the timing properties of PSDL operators. We consider schedulers for PSDL programs containing only periodic operators. We define a task to be the instance of operator occurred in a particular period. A constraint graph of PSDL operators is obtained by building a harmonic block, evaluating the number of tasks in the graph, generating the chains of tasks, and interconnecting these chains.

Known optimal algorithms for non-preemptive multiprocessor scheduling are very time consuming. We propose two fast greedy methods based on different heuristic cost functions.

Given a constraint graph, we propose an Earliest Task Deadline/Latest Processor Ending Time First algorithm called Algorithm A. The reason to assign earliest deadline task with highest priority is to ensure that task can meet its deadline as well as possible. This is a fast heuristic algorithm that does not guarantee a feasible solution.

We also propose an Earliest Task Start Time/Most Available Processor First algorithm called Algorithm B. The key idea in this algorithm is that tasks are first-ready, first-served.

We create two methods to search for the optimal solution. The first method traverses each possible solution and finds the best one. The second method uses a lower bound cost estimate to limit the search, following the branch and bound approach. This method is much better than the first one because it can save time and space due to unnecessary graph expanding.

B. FURTHER RESEARCH

This is the first work on scheduling PSDL operators on multiprocessor systems. Further research is required for implementation and identification of possible weaknesses. Because the tasks are statically scheduled, it is difficult to identify all possible software design contingencies without an executable version of the scheduler. The author recommends continued work in the following areas:

Implementation of the Static Scheduler,

Implementation of the Execution Support System Interfaces,

Modifying proposed algorithms using better heuristics,

Proving the algorithms by mathematics, and

Changing the assumptions of scheduling problem.

1. Implementation of the Static Scheduler

The programming language used in implementation will be Ada. The guides for accomplishing the implementation are contained in [Ref. 19] and [Ref. 33].

2. Implementation of the Execution Support System Interfaces

Section C of Chapter one briefly introduces the relationship between the three components of the Execution Support System. In Figure 6, the static scheduler interfaces with the dynamic scheduler. Since the algorithms for both schedulers were designed independently, there may need to be some modifications made to ensure proper execution. The static scheduler passes a separate text

file to the dynamic scheduler containing information about the non-time critical operators in a prototype. There should be an interface between these two schedulers, which indicates the format for communication between both schedulers. This is an implementation problem rather than a design problem. More detail can be found in [Ref. 35: pp. 49-50].

3. Modifying Proposed Algorithms Using Better Heuristics

Any heuristic method suffers from several shortcomings such as the difficulty in assuring its solution accuracies [Ref. 21]. If one algorithm can be proven to be better than the other which had been proven to be optimal, then we can also say that this algorithm is also optimal. The rule of thumb can be applied to all scientific inventions, including scheduling problems. The more basic understanding of the problems we have, the more opportunities we can invent a heuristic solution.

4. Proving the Algorithms by Mathematics

We can also build a mathematical model for the proof of algorithm. Although some heuristic algorithms seem to be very straightforward, it is sometimes hard to completely prove that they work correctly.

5. Changing the Assumptions of Scheduling Problem

Different assumptions can lead to different results. For instance, tasks are assumed to be non-preemptive in this thesis, but they could be preemptive. There are still many open problems to be

discussed, such as periodic or non-periodic, any constraints or not, whether the constraints graph is a tree or network, scheduling tasks on multiprocessor systems or on distributed systems, etc.

C. CONCLUSIONS

The goal of this thesis is to design the algorithms to schedule tasks in multiprocessor systems. There are two kind of algorithms created. One of them is under the assumption that tasks have no timing constraints on them, and the other have.

One important aspect of the multiprocessor is its application in real-time systems. Computer architecture had made rapid progress in the manufacture of chips. This makes processors cheaper than before. Progress is now limited by software problems. Scheduling problems are among these software problems. They must be solved to arrange tasks so that we can utilize multiprocessors to the maximum.

LIST OF REFERENCE

1. [BB88] Brassard, G., Bratley, P., Algorithmics, Prentice-Hall, Englewood Cliffs, NJ., 1988.

2. [Ber89] Berzins, V., Course Notes - CS4500, Naval Postgraduate School, Monterey, CA., 1989.

3. [Boo87] Booch, G., Software Engineering with Ada, 2nd ed., Benjamin/Cummings Publishing Co., Inc., Menlo Park, CA., 1987.

4. [Bro89] Brookshear, J. G., Theory of Computation, Benjamin/Cummings Publishing Co., Inc., Menlo Park, CA., 1989.

5. [Cer89] Cervantes, J. J., An Optimal Static Scheduling Algorithm for Hard Real-time Systems. M.S. thesis, Naval Postgraduate School, Monterey, CA., Dec. 1989.

6. [Cof76] Coffman, E. G., Computer and Job-Shop Scheduling Theory, John Wiley & Sons, Inc., 1976.

7. [Coo83] Cook, S. A., "An Overview of Computational Complexity", Communications of the ACM, Vol. 26, No. 6, Jun. 1983, pp. 401-408.

8. [CSR87] Cheng, S. C., Stankovic, J. A., Ramamritham, K., "Scheduling Algorithms for Hard Real-time Systems - A Brief Survey", COINS Technical Report 87-55, Jun. 10, 1987.

9. [DOSE87] Dossey, J. A., Otto, A. D., Spence, L. E., Eynden, C. V., Discrete Mathematics, Scott, Foresman and Company, 1987.

10. [FB73] Fernandez, E. B., Bussell, B., "Bounds on the Number of Processors and Time for Multiprocessor Optimal Schedules", IEEE Transactions on Computers, Vol. C-22, No. 8, Aug. 1973, pp. 745-751.

11. [Fre82] French, S., Sequencing and Scheduling, Ellis Horwood Ltd., 1982.

12. [GGK78] Garey, M. R., Graham, R. L., Johnson, D. S., "Performance Guarantees for Scheduling Algorithms", Operations Research, Vol. 26, No. 1, Jan.-Feb., 1978, pp. 3-21.

13. [GJ78] Garey, M. R., Johnson, D. S., ""Strong" NP-Completeness Results: Motivation, Examples, and Implications", Journal of the Association for Computing Machinery, Vol. 25, No. 3, Jul. 1978, pp. 499-508.

14. [GJ79] Garey, M. R., Johnson, D. S., Computers and Intractability; A guide to the Theory of NP-Completeness, Freeman; San Francisco, 1979.

15. [GM84] Gondran M., Minoux, M., Graphs and Algorithms, John Wiley & Sons Ltd., 1984.

16. [Gra69] Graham, R. L., "Bounds on Multiprocessing Timing Anomalies", SIAM J. Appl. Math. Vol. 17, No. 2, Mar. 1969, pp. 416-429.

17. [HS78] Horowitz, E., Sahni, S., Fundamentals of Computer Algorithms, Computer Science Press, Rockville, MD., 1978.

18. [Hu61] Hu, T. C., "Parallel Sequencing and Assembly Line Problems", Operations Research, Vol. 9, Nov. 1961, pp. 841-848.

19. [Jan88] Janson, D. M., A Static Scheduler for The Computer Aided Prototyping System: An Implementation Guide, M.S. thesis, Naval Postgraduate School, Monterey, CA., Sep. 1988.

20. [Kil89] Kilic, M., Static Schedulers for Embedded Real-time Systems. M.S. thesis, Naval Postgraduate School, Monterey, CA., 1989.

21. [KN84] Kasahara, H., Narita, S., "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing", IEEE Transactions on Computers, Vol. c-33, No. 11, Nov. 1984, pp. 1023-1029.

22. [LB88] Luqi, Berzins, V., "Rapidly Prototyping Real-time Systems", IEEE Software, Sep. 1988, pp. 25-36.

23. [LBY88] Luqi, Berzins, V., Yeh, R. T., "A Prototyping Language for Real-time Software", IEEE Transactions on Software Engineering, Vol. 14, No. 10, Oct. 1988, pp. 1409-1423.

24. [LK78] Lenstra, J. K., Kan, A. H. G. R., "Complexity of Scheduling under Precedence Constraints", Operations Research, Vol. 26, No. 1, Jan.-Feb. 1978, pp. 22-35.

25. [LK88] Luqi, Ketabchi, M., "A Computer-Aided Prototyping System", IEEE Software, Mar. 1988, pp. 66-72.

26. [Luq88] Luqi, "Knowledge-Based Support for Rapid Software Prototyping", IEEE Expert, Winter 1988, pp. 9-18.

27. [Luq89a] Luqi, "Software Evolution Through Rapid Prototyping", IEEE Computer, May 1989, pp. 13-25.

28. [Luq89b] Luqi, "Rapid Prototyping Languages and Expert Systems", IEEE Expert, Summer 1989, pp. 2-5.

29. [Luq89c] Luqi, "Handling Timing Constraints in Rapid Prototyping", *in proceedings of the* 22nd Annual Hawaii International Conference on System Science, Kailua-Kona, Hawaii, Jan. 1989.

30. [LW66] Lawler, E. L., Wood, D. E., "Branch-and-Bound Methods: A Survey", Operations Research, Vol. 14, Jul. 1966, pp. 699-719.

31. [Man67] Manacher, G. K., "Production and Stabilization of Real-Time Task Schedules", Journal of the Association for Computing Machinery, Vol. 14, No. 3, Jul. 1967, pp. 439-465.

32. [Man89] Manber, U., Introduction to Algorithms, Addison-Wesley Publishing Company Inc., 1989.

33. [Mar88] Marlowe, L., A Scheduler for Critical Time Constraints, M.S. thesis, Naval Postgraduate School, Monterey, CA., Dec. 1988.

34. [MC69] Munitz, R. R., Coffman, E. G. JR., "Optimal Preemptive Scheduling on Two_Processor Systems", IEEE Transactions on Computers, Vol. c-13, No. 11, Nov. 1969, pp.1014-1020.

35. [O'He88] O'Hern, J. T., A Conceptual Level Design for a Static Scheduler for Hard Real-time Systems, M.S. thesis, Naval Postgraduate School, Monterey, CA., Sep. 1988.

36. [Pre87] Pressman, R. S., Software Engineering, 2nd ed., McGraw-Hill, Inc., 1987.

37. [RCG72] Ramamoorthy, C. V., Chandy, K. M., Gonzalez, JR., "Optimal Scheduling Strategies in a Multiprocessor System", IEEE Transactions on Computers, Vol. c-21, No. 2, Feb. 1972, pp. 137-146.

38. [SH78] Sahni, S., Horowitz, E., "Combinatorial Problems: Reducibility and Approximation", Operations Research, Vol. 26, No. 5, Sep.-Oct. 1978, pp. 718-759.

39. [SR88] Stankovic J. A., Ramamritham, K., Hard Real-Time Systems, IEEE Computer Society Press, Washington, DC., 1988.

40. [Wei77] Weide, B., "A Survey of Analysis Techniques for Discrete Algorithms", Computing Surveys, Vol. 9, No. 4, Dec. 1977, pp. 291-313.

41. [ZRS87] Zhao, W., Ramamritham, K., Stankovic, J. A., "Preemptive Scheduling Under Time and Resource Constraints", IEEE Transactions on Computers, Vol. C-36, No. 8, Aug. 1987, pp. 949-960.

42. [CR75] Chandy, K. M., Reynolds, P. F., "Scheduling partially ordered tasks with probabilistic execution times", in Proceedings of the 5th Symposium on Operating Systems Principles, 1975.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA 93943	1
Office of Naval Research 800 N. Quincy Street Arlington, VA 22217-5000	1
Center for Naval Analysis 4401 Ford Avenue Alexandria, VA 22302-0268	1
National Science Foundation ATTN: Dr. K. C. Tai Division of Computer and Computation Research Washington, D.C. 20550	1
Office of the Chief of Naval Operations Code OP-941 Washington, D.C. 20350-2000	1
Office of the Chief of Naval Operations Dr. John R. Davis, Chief Scientist Code OP-094H Washington, D.C. 20350-2000	1

Office of the Chief of Naval Operations 1
Code OP-945
Washington, D.C. 20350-2000

Office of the Chief of Naval Operations 1
ATTN: CDR R. V. Gragg
Code OP-942F4
Washington, D.C. 20350-2000

Commander 2
Naval Sea Systems Command
ATTN: LCDR Scott Kelly
Code 06D3131
Washington, D.C. 20362-5101

Commander 2
Naval Telecommunications Command
Naval Telecommunications Command Headquarters
4401 Massachusetts Avenue NW
Washington, D.C. 20390-5290

Commander 1
Naval Data Automation Command
Washington Navy Yard
Washington, D.C. 20374-1662

Commander 2
Space and Naval Warfare Systems Command
ATTN: CAPT John Gauss
PMW-162
Washington, D.C. 20363-5100

Assistant Secretary of the Navy 1
Research, Development and Acquisitions
Washington, D.C. 20350-1000

Dr. Lui Sha 1
Carnegie Mellon University
Software Engineering Institute
Department of Computer Science
Pittsburgh, PA 15260

Commanding Officer 1
Code 5150
Naval Research Laboratory
Washington, D.C. 20375-5000

Defense Advanced Research Projects Agency (DARPA) 1
Director, Naval Technology Office
1400 Wilson Boulevard
Arlington, VA 2209-2308

Defense Advanced Research Projects Agency (DARPA) 1
Director, Prototype Projects Office
1400 Wilson Boulevard
Arlington, VA 2209-2308

Defense Advanced Research Projects Agency (DARPA) 1
Director, Tactical Technology Office
1400 Wilson Boulevard
Arlington, VA 2209-2308

Dr. R. M. Carroll (OP-01B2) 1
Chief of Naval Operations1
Washington, DC 20350

Dr. Robert M. Balzer 1
USC-Information Sciences Institute
4676 Admiralty Way
Suite 1001
Marina del Ray, CA 90292-6695

Dr. Ted Lewis 1
Editor-in-Chief, IEEE Software
Oregon State University
Computer Science Department
Corvallis, OR 97331

Dr. R. T. Yeh
International Software Systems Inc. 1
12710 Research Boulevard, Suite 301
Austin, TX 78759

Dr. C. Green 1
Kestrel Institute
1801 Page Mill Road
Palo Alto, CA 94304

Prof. D. Berry 1
Department of Computer Science
University of California
Los Angelas, CA 90024

Director, Naval Telecommunications System Integration Center 1
NAVCOMMUNIT Washington
Washington, D.C. 20363-5110

Dr. Knudsen 1
Code PD50
Space and Naval Warfare Systems Command
Washington, D.C. 20363-5110

Ada Joint Program Office 1
OUSDRE(R&AT)
The Pentagon
Washington, D.C. 23030

CAPT A. Thompson 1
Naval Sea Systems Command
National Center #2, Suite 7N06
Washington, D.C. 22202

Dr. Peter Ng 1
New Jersey Institute of Technology
Computer Science Department
Newark, NJ 07102

Dr. Van Tilborg 1
Office of Naval Research
Computer Science Division, Code 1133
800 N. Quincy Street
Arlington, VA 22217-5000

Dr. R. Wachter 1
Office of Naval Research
Computer Science Division, Code 1133
800 N. Quincy Street
Arlington, VA 22217-5000

Dr. J. Smith, Code 1211 1
Office of Naval Research1
Applied Mathematics and Computer Science
800 N. Quincy Street
Arlington, VA 22217-5000

Mr. William E. Rzepka 1
U.S. Air Force Systems Command
Rome Air Development Center
RADC/COE
Griffis Air Force Base, NY 13441-5700

Dr. C.V. Ramamoorthy 1
University of California at Berkeley
Department of Electrical Engineering and Computer Science
Computer Science Division
Berkeley, CA 90024

Dr. Nancy Levenson 1
University of California at Irvine
Department of Computer and Information Science
Irvine, CA 92717

Mr. George Roberson 1
Fleet Combat Direction Systems Support Activity
San Diego, CA 92147-5081

Mr. William Hanley 1
Fleet Combat Direction Systems Support Activity
San Diego, CA 92147-5081

Dr. Earl Chavis (OP-162) 1
Chief of Naval Operations
Washington, DC 20350

Dr. Alan Hevner 1
University of Maryland
College of Business Management
Tydings Hall, Room 0137
College Park, MD 20742

Dr. Al Mok 1
University of Texas at Austin
Computer Science Department
Austin, TX 78712

George Sumiall 1
US Army Headquarters
CECOM
AMSEL-RD-SE-AST-SE
Fort Monmouth, NJ 07703-5000

Mr. Joel Trimble 1
1211 South Fern Street, C107
Arlington, VA 22202

Linwood Sutton 1
Code 423
Naval Ocean Systems Center
San Diego, CA 92152-5000

Dr. Sherman Gee 1
Code 221
Office of Naval Technology
200 N. Quincy St.
Arlington, VA 22217

Dr. Mark Kellner 1
Carnegie-Mellon University
Software Engineering Institute
Pittsburgh, PA 15213

Dr. Luqi 1
Code 52Lq
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Dr. Valdis Berzins 30
Code 52Bz
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Dr. Man-Tak Shing 1
Code 52Sh
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Commanding Officer 1
Naval Command and Control Systems Command
Wibbelhof St. 3
2340 Wilhelmshaven
West Germany

Commander 1
Naval Ocean Systems Center
Code 43
San Diego, CA 92152-5000
ATTN: CDR Wayne Duke

Commander 1
Naval Ocean Systems Center
Code 44
San Diego, CA 92152-5000
ATTN: Dr. Glenn Osga

Dr. Chuck Hutchins 1
Code 55Hu
Operations Research Department
Naval Postgraduate School
Monterey, CA 93943

Dr. Gary Poock 1
Code 55Pk
Operations Research Department
Naval Postgraduate School
Monterey, CA 93943

LCDR James A. Seveney 2
8507 Shirley Woods Ct.
Lorton, VA 22079

Commanding Officer 2
1st Destroyer Squadron
Mecklenburger St. 50 A I

Superintendent
Chung-Cheng Institute of Technology
Tao-Yuan 33500, Taiwan, ROC
1

Library of Chung-Cheng Inst. of Tech.
Chung-Cheng Institute of Technology
Tao-Yuan 33500, Taiwan, ROC

Chairman Kuo
Department of Vehicle Engineering
Chung-Cheng Institute of Technology
Tao-Yuan 33500, Taiwan, ROC
1

Chairman
Department of Information Science
Chung-Cheng Institute of Technology
Tao-Yuan 33500, Taiwan, ROC
1

Chairman
Computer Center
Chung-Cheng Institute of Technology
Tao-Yuan 33500, Taiwan, ROC
1

Liangchuan Hsu
229-12 Chung-hsiao Rd., Ping-tung,
Taiwan, ROC
2